

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

JavaScript Frameworks
for Modern Web Dev

异步图书
www.epubit.com.cn

Apress®

JavaScript 开发框架权威指南

[美] Tim Ambler Nicholas Cloud 著
一心一译前端小组 译

• 现代Web开发不可或缺的18种JavaScript库和框架的详尽指南



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

JavaScript Frameworks
for Modern Web Dev

JavaScript 开发框架权威指南

[美] Tim Ambler Nicholas Cloud 著
一心一译前端小组 译

人民邮电出版社
北京

图书在版编目(CIP)数据

JavaScript开发框架权威指南 / (美) 安布勒
(Tim Ambler), (美) 克劳德 (Nicholas Cloud) 著;
一心一译前端小组译. — 北京: 人民邮电出版社,
2017.5

ISBN 978-7-115-44719-7

I. ①J… II. ①安… ②克… ③一… III. ①JAVA语
言—程序设计—指南 IV. ①TP312.8-62

中国版本图书馆CIP数据核字(2017)第048275号

版权声明

JavaScript Frameworks for Modern Web Development

by Tim Ambler and Nicholas Cloud ISBN: 978-1-4842-0662-1

Original English language edition published by Apress Media.

Copyright © 2015 by Apress Media.

Simplified Chinese-language edition copyright ©2017 by Post & Telecom Press

All rights reserved.

本书中文简体字版由 Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

-
- ◆ 著 [美] Tim Ambler Nicholas Cloud
 - 译 一心一译前端小组
 - 责任编辑 陈冀康
 - 责任印制 焦志炜
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 三河市海波印务有限公司印刷
 - ◆ 开本: 800×1000 1/16
 - 印张: 25.25
 - 字数: 605千字 2017年5月第1版
 - 印数: 1-2500册 2017年5月河北第1次印刷

著作权合同登记号 图字: 01-2016-0777 号

定价: 89.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316

反盗版热线: (010)81055315

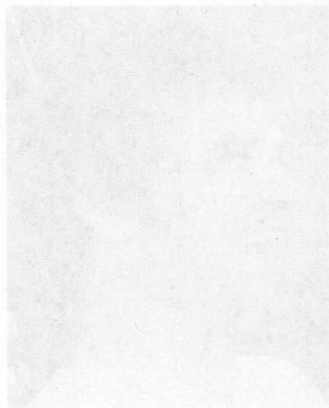
广告经营许可证: 京东工商广字第8052号

内容提要

JavaScript 是现代 Web 开发必不可少的编程语言，但 JavaScript 的生态系统包括库、框架以及工具都在快速地发展且日益庞大。程序员学习的需求和面临的挑战也相应地增加。

本书涵盖了在开发过程中常用的各种 JavaScript 工具，以帮助读者在大量流行的 JavaScript 工具中做选择。全书分为 16 章，从开发工具、模块加载器、客户端框架、服务端框架数据库交互、通信、管理控制流和其他有用框架等几个方面，涵盖了 Bower、Grunt、Yeoman、PM2、RequireJS、Browserify、Knockout、AngularJS、Kraken、Mach、Mongoose、Knex、Bookshelf、Faye、Q、Async.js、Underscore 和 Lodash 等框架和库。全书涵盖了客户端和服务端端的开发，通过细致的讲解、详细的代码示例，阐明了这些工具的用法。

本书主要面向熟悉 JavaScript 但对数量庞大的解决方案感到困惑的开发者，对于想要学习 JavaScript 并掌握各种相关工具的读者来说，具有很好的参考价值。

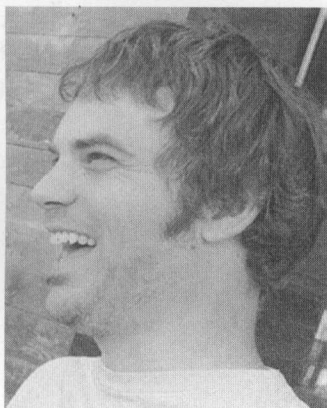


作者简介

ISBN 978-7-115-44719-7



Tim Ambler 是来自美国田纳西州那什维尔的一名软件工程师。他对编程的热情来自于父亲。在他小的时候，他的父亲就向他介绍了 Commodore 64 电脑。Tim 是几个流行的开源项目的作者，其中 whenLive 已经被 GitHub 员工采用。作为会议演说家和多产的作家，Tim 多次被在线出版物推荐，如 JavaScript Weekly 和 Node Weekly。Tim 目前与他的妻子和 2 只猫生活在南部。读者可以在 Twitter 上 @tkambler 关注他。



Nicholas Cloud 是一名软件工程师，居住在非常潮湿的城市圣路易斯。过去十几年里，他利用自己的技能成就一番成功事业。通过 JavaScript、C# 和 PHP，他开发了大量适用于多终端的 Web 应用、Web 服务、桌面应用。Nicholas 是开源软件的有力支持者，致力于用户级项目开发，并写了几个自己的开源库。在业余时间，他在不同的用户组发言、参加会议、写书、写技术文章、写博客。他的 Twitter 是 @nicholascloud。

网址: <http://www.apress.com.cn>

河南高校印务有限公司印刷

★ 开本: 786×1092 1/16

印张: 22.25

字数: 500 千字

印数: 1-2 500 册

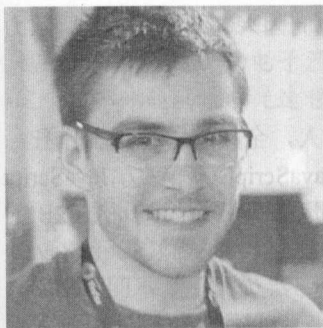
定价: 89.00 元

读者服务热线: (010) 81055410 邮购热线: (010) 81055414

反盗版热线: (010) 81055414

经营许可证: 京新证字(2015)第 0052 号

技术评阅者简介



Robin Hawkes 致力于学习并热衷于结合设计与编码来解决问题。他是 *Foundation HTML 5 Canvas* (Apress, 2011) 一书的作者，这本书是关于使用 JavaScript 开发游戏的。他也是 ViziCities one-man 乐队的成员，ViziCities 是一个 WebGLS-3D 城市虚拟平台。过去 Robin 一直致力于 Mozilla 和 Pusher 的全球开发者关系。

JavaScript 社区中正在迈开近乎狂热的创新步伐，虽然充满了无穷的魅力，但也提出了独特的挑战。JavaScript 的生态系统包括库、框架以及工具，都在激烈地成长。过去针对问题可能只有少量的解决方案，而今已有成千上万种。随着工具库的快速增长，开发人员要面对这样的艰难任务：在很多看起来不错的方案中选择合适工具。

JavaScript 是一门原本设计用于解决相当简单问题的语言，现在却以最初没有预见的速度成长。更重要的是，JavaScript 是一门优美的表达语言，但它不是没有棱角和潜在的陷阱。尽管它看起来简单，但要掌握它并不容易。JavaScript 是一门优美的表达语言，但它不是没有棱角和潜在的陷阱。尽管它看起来简单，但要掌握它并不容易。JavaScript 是一门优美的表达语言，但它不是没有棱角和潜在的陷阱。尽管它看起来简单，但要掌握它并不容易。

由于这些和很多其他原因，开发社区的很大一部分人都在探究如何最好地应用 JavaScript。我们编写了本书，希望能够帮助您在大量流行的 JavaScript 工具中被选择。这些工具解决开发中遇到的问题，从简单的任务到复杂的任务，从简单的任务到复杂的任务，从简单的任务到复杂的任务。

我们编写了本书，希望能够帮助您在大量流行的 JavaScript 工具中被选择。这些工具解决开发中遇到的问题，从简单的任务到复杂的任务，从简单的任务到复杂的任务，从简单的任务到复杂的任务。我们编写了本书，希望能够帮助您在大量流行的 JavaScript 工具中被选择。这些工具解决开发中遇到的问题，从简单的任务到复杂的任务，从简单的任务到复杂的任务。

我们编写了本书，希望能够帮助您在大量流行的 JavaScript 工具中被选择。这些工具解决开发中遇到的问题，从简单的任务到复杂的任务，从简单的任务到复杂的任务，从简单的任务到复杂的任务。我们编写了本书，希望能够帮助您在大量流行的 JavaScript 工具中被选择。这些工具解决开发中遇到的问题，从简单的任务到复杂的任务，从简单的任务到复杂的任务。

在写这本书时，我们曾尝试对围绕 JavaScript 的“喧嚣不休的洪流”做筛选。这样做能让我们相信，我们的读者会听到。我们希望这些文章能像对我们一样，也对您有所帮助。

译者简介

一心一译前端小组由一群热爱前端和 JavaScript 的技术人员组成，他们因兴趣而走到一起，致力于为读者奉献高品质的翻译技术图书。这个前端小组包括以下成员：

郭凯

美团点评酒旅事业群前端团队负责人，高级技术专家，资深互联网人，全栈工程师，工作狂，崇尚工匠精神，曾就职于音悦台、淘宝旅行。翻译作品有《编写可维护的 JavaScript》《第三方 JavaScript 编程》，有 In、Juicer、jQuery、F2E.im、PM25 等开源项目，业余时间负责开源前端技术社区 F2E 的开发和维护。

王思可

热爱前端开发，热爱技术。曾在阿里旅行工作，现在供职于灵雀云，希望能够更好地用技术服务于人。

朱佳慧

美团点评酒旅事业群前端开发工程师，2015 年加入美团。

赵荣娇

花名银翘，前端开发工程师，也是《响应式设计、改造与优化》一书的译者。著有《超实用的 CSS 代码段》《代码逆袭：超实用的 HTML 代码段》等图书。喜欢旅行，热爱前端开发。正在学习如何成为一名胜过好老师的妈妈。

马荃

美团点评酒旅事业群高级前端工程师，曾就职于创新工场和阿里巴巴，擅长前端工具自动化，关注前端体验和效率，乐于分享。

巩守强

猫眼基础交易负责人，美团前端高级技术专家，对于前端工程化、前端性能优化和客户端动态化感兴趣。

蔡平

曾在美团酒旅前端和 360 奇舞团工作，现任海致前端开发 Leader。喜欢前端技术，关注前端最新动态，追求高效的前端工程化解决方案。

康明轩

北京师范大学硕士，对世界充满好奇的孩子，迷恋于技术不能自拔的瘾士，掉在麻袋里的书虫。走一条从修地球到修电脑的非典型路线，现就职于北京指南针科技发展股份有限公司，从事网络应用开发，认为程序员也是一门可以长相厮守的手艺。

前言

人们说我们生活在一个信息时代，但似乎没有一条是我所需要的或想了解的信息。事实上，（我越来越相信）这一切电子产品只会增加我们的困惑，发表内部独家新闻、裁决几乎还没开始的事件：这喋喋不休的洪流以光速移动，以至于几乎不可能听到重要的事。

——马修·弗拉曼 《The Kingdom of Ohio》

“技术发展迅速”是一条老生常谈的格言，而且有很好的理由：技术的确发展迅速。但此时，JavaScript 的确发展得特别迅速——就像马修·弗拉曼在俄亥俄王国所说的“喋喋不休的洪流以光速移动”。随着基于浏览器应用迅速增长的复杂度以及服务器端 JavaScript 的日益普及，这门语言正在经历被许多人所谓的复兴之中。一切多亏了 Node.js。

JavaScript 社区中正在迈开近乎狂热的创新步伐，虽然充满了无穷的魅力，但也提出了自己的独特的挑战。JavaScript 的生态系统包括库、框架以及工具，都在剧烈地成长。过去针对任何给定问题可能只有少量的解决方案，而今已经有许多解决方案可以选择，并且其数目日益增长。因此，开发人员要面对这样的艰难任务：在很多看起来不错的方案中选择合适工具。

如果你像我们一样好奇为什么最近 JavaScript 似乎吸引了如此多的注意力，那么值得停下来思考 JavaScript 的本质。这门语言由一个人在十天内创造，现在却作为许多我们知道 Web 站点的基础服务。一门原本设计用于解决相当简单问题的语言，现在却以最初没有预见到的创新方式所应用。更重要的是，JavaScript 是一门优美的表达语言，但它不是没有棱角和潜在的陷阱。尽管它灵活、高效以及无处不在，但是对于 JavaScript 的初学者来说，了解 JavaScript 的一些概念如事件循环、原型继承等，是特别具有挑战性的。

由于这些和很多其他原因，开发社区的很大一部分人都在探究如何最好地应用 JavaScript 的独特特性。毫无疑问，我们只是抓住了语言的表面和其背后社区提供的能力。对于那些对知识有着贪婪的需求和充满创造欲的人，现在是成为一名 JavaScript 开发者的最佳时机。

我们编写了本书来指导你在大量流行的 JavaScript 工具中做选择，这些工具解决开发技术栈的两端：浏览器和服务端。教程及本书中可下载的代码示例阐明了这些工具的用法，包括依赖管理、模块化代码模式、自动化重复任务构建、创建专业的服务、客户端应用的架构、灵活的水平扩展、执行事件日志记录和与不同的数据存储交互。

本书涵盖的库和框架包括 Bower、Grunt、Yeoman、PM2、RequireJS、Browserify、Knockout、AngularJS、Kraken、Mach、Mongoose、Knex、Bookshelf、Faye、Q、Async.js、Underscore 和 Lodash。

在写这本书时，我们的目标是对围绕 JavaScript 的“喋喋不休的洪流”做筛选。这样做能让我们相信，的确有重要的事情需要被听到。我们希望这些文字能像对我们一样，也对您有所帮助。

本书读者对象

本书主要面向能胜任 JavaScript 的，但又对庞大数量、似乎能解决所有问题的方案感到困惑的开发者。本书帮你驱散迷雾，为你提供一个深入的指导，引导你学习知名组织现在正在使用并取得了很大成功的库和框架。本书话题涵盖客户端和服务端开发。如果你至少对浏览器文档对象模型 (DOM)、常用的客户端库（类似 jQuery 和 Node.js）有所熟悉，那么你会从本书中获益。

本书模式

本书涵盖了在整个开发过程中能用到的 JavaScript 工具，从项目的第一次代码提交到发布甚至除此以外的问题，为你提供了多种选择。为此，本书的章节可分为以下部分。

第 1 部分：开发工具

Bower

依赖管理已经不是一个新想法了——著名的例子包括 Node 的 npm、Python 的 pip 和 PHP 的 composer。然而有一种实践方案直到最近才被广泛采用，这个概念是管理 Web 应用前端资源——JavaScript 库、样式表、字体、图标和图像，以此作为构建现代 Web 应用的砖瓦。在这一章，你会学习几种管理方式。此领域的一个流行工具 Bower，可以为你提供一个机制来组织应用程序中的依赖项，改进你的开发过程。

Grunt

Perl 语言的创造者拉里·沃尔，描述了伟大程序员的三种特质：懒惰，缺乏耐心，以及傲慢。这一章中，会集中讨论帮你加强这三种特质中“懒惰”的工具——Grunt。这个流行的任务处理器为开发者提供了创建命令行工具的框架。它能自动构建重复任务，例如运行测试脚本、拼接文件、编译 SASS/LESS 样式表、检查 JavaScript 代码错误等。阅读完这一章，你会了解如何使用几个流行的 Grunt 插件，以及如何创建和在社区中分享自己的插件。

Yeoman

Yeoman 为开发者提供了创建可重用模板（“生成器”）的机制，通过其模板描述项目文件模式（项目初始化所需要的依赖，Grunt 任务等），这样一来就能轻易做到反复重用。社区的广泛支持让你可以利用大量现存的模板。这一章中，会详细讨论安装 Yeoman 以及如何使用一些现存的生成器。随后，会讲解怎样创建和在社区中分享自己的模板。

PM2

本章中，会通过学习 PM2 来结束关于开发工具议题的讨论。PM2 是一个命令行工具，它简化了很多与运行 Node 应用相关的任务，包括监视应用状态以及为满足增长的需求而高效地伸缩应用。

第 2 部分：模块加载器

RequireJS 和 Browserify

JavaScript 对于在浏览器中加载外部依赖先天缺乏的特点，令开发者很沮丧。好在社区使用两种

不同和相互竞争的标准填补了这一空缺：异步模块加载（AMD）API 和 CommonJS。我们会深入探索两者的细节并看一看两者广泛使用的实现方式：RequireJS 和 Browserify。每一个都有其优点，我们都会详细探讨，但两者都对构建应用程序有深刻的积极影响。

第 3 部分：客户端框架

Knockout 和 AngularJS

近年来，Web 开发人员见证了所谓的“单页面应用程序的迅速流行”。具有这种行为的应用曾经只在桌面上可用，在浏览器中增加了代码的复杂度。这一节中，深入介绍两种广泛使用的前端框架：Knockout 和 AngularJS。它们提供了已经被证明的模式，能帮你解决频繁遇到的问题，减少代码的复杂性。Knockout 聚焦于解决视图和数据之间的关系，但另一方面也让开发者自由判定应用程序架构。AngularJS 则提供了一个更规范的方法，涵盖了视图、应用路由、数据传输和模块设计。

第 4 部分：服务端框架

Kraken 和 Mach

在没有服务端交互的情况下，客户端应用程序不会很有用。在本部分中，看一看两个流行的框架：Kraken 和 Mach。它们支持开发人员创建后端应用程序。Mach 不仅仅是一个简单的 Web 服务器，它是 Web 的 HTTP 层。Mach 既能够通过一个直观的、可扩展的 HTTP 栈来提供内容，又能获取内容。Mach 的接口包含了 Node.js 应用中 Web 页面请求的服务，也包括在浏览器中用 Mach 的 AJAX 请求获取 JSON 数据，甚至包括向其他网络堆栈请求的完全重写及代理。在许多方面，可以说 Mach 是 HTTP 层的瑞士军刀。

第 5 部分：管理数据库交互

Mongoose、Knex 和 Bookshelf

每个应用程序的核心都存在着任何开发栈中最重要的组成部分——用户搜索的数据。在本部分中，熟悉两个库，它们帮助你简化与流行的存储平台（如 MongoDB、MySQL、PostgreSQL 和 SQLite）交互的一些复杂性。读完这一节，你就能轻松地定义模式、关联、生命周期“钩子”等。

第 6 部分：通信

Faye

本部分中，介绍 Node.js 库 Faye。它为开发人员提供了一个健壮的、易于使用的平台，依靠服务器和所有主流浏览器之间的实时通信来构建产品。Faye 之所以流行，源于 Web 项目有着希望在各处运行的目标。Faye 通过提供无缝的回调支持多种通信协议来完成这个目标。

第 7 部分：管理控制流

Q 和 Async.js

JavaScript 异步的特性给开发者很大的灵活度——与强迫开发者以线性方式执行代码相反，

JavaScript 允许开发者同时协调多个行为。不幸的是，伴随着这种灵活性的是额外相当程度的复杂性——也就是许多开发人员所说的“地狱回调”或“噩梦金字塔”。在本部分中，考察两种流行的库：Q 和 Async.js，它们将帮助你解决异步控制流的复杂性问题。

第 8 部分：其他有用的库

对一些其他非常有用的库，本书将忽略，不做涉及。但有些额外的库并不是不值一提。这一部分将介绍这些库。

Underscore 和 Lodash

Underscore（以及它的继承者 Lodash）是一个相当有用的函数集合，能简化很多频繁使用但实现起来冗长的模式。本章简要介绍了这些库，还提到了一些更流行的扩展，它们能用来进一步提高其效用。示例代码强调了这些库中使用最频繁的一部分。

源代码下载

本书的每个章节都包含很多示例，源代码的 zip 版本可以从 <http://www.apress.com/9781484206638> 或 www.epubit.com.cn 下载。zip 文件的子目录包含的源码对应每一章节中的示例。每章示例中源码的第一行注释都标识了源代码所在的文件路径。例如，如果你读到了第 10 章（Mach）的清单 10-1，实际的源码文件放在 `mach/example-000/no-such-file.js` 路径下，相对于提取后的 `mach.zip` 文件路径。

清单 10-1 一个假想的例子

```
// example-000/no-such-file.js
console.log('this is not a real example');
```

大部分示例在 Node.js 环境运行，Node.js 可以从 <https://nodejs.org> 下载。对于如何下载和安装每章中的示例，相应章节都会提前做相关知识解释。（例如，阐述 Mongoose 的第 11 章中必须运行 MongoDB）

任何关于运行示例代码的额外步骤（如执行 curl 请求）或与一个正在运行中的示例进行交互（如打开 Web 浏览器并导航到指定 URL）都会在清单旁做出解释。

第 2 部分：模块加载器

RequireJS 和 Browserify

致谢

本书的诞生和你们的鼓励与支持息息相关：

Nicholas Cloud，我的朋友及合作者。正是因为他，这本书才能够保持深度和广度。他的学识、经验以及坚定致力于项目的精神带来了不可度量的帮助。在此表示感谢。

Louise Corrigan、Kevin Walter、Christine Ricketts、Melissa Maldonado 和其他 Apress 的员工，在这本书的写作过程中给予我们支持。很感谢他们邀请我来创作并持续提供支持。

技术评审 Robin Hawkes，本书的示例和源代码得益于他们的洞察力和犀利的眼光。

Faye 的作者 James Coglan，谢谢你分享专业技术知识和反馈。

我的朋友及同事 Greg Jones、Jeff Crump、Seth Steele、Jon Zumbrun 和 Brian Hiatt，非常感谢你们的反馈和鼓励。

——Tim

感谢是很无力的东西，还有很多感激未能言表：

首先，感谢我的合作者 Tim，感谢他邀请我参与本书的创作。我们从未谋面，半年以来，我们通过远程合作办公，有足够的时间给彼此留下深刻的印象。非常感谢他对我信任、鼓励和持续的努力。

其次，感谢 Apress 的员工：Kevin、Louise、Christine 和 Melissa，他们指导我整个出版的过程。毫不夸张地说，他们的耐心和细致引导，使我在整个写作过程中脚踏实地。他们都是顶尖的专业人士，期待有一天能与之共事。

再次，我感谢非常优秀的技术评审 Robin，比开发人员更加负责地阅读并执行代码。

最后，我不能偿还当研究 Mach、Knockout 这些主题知识时从 Michael Jackson (@mjackson) 和 Ryan Niemeyer (@RPNiemeyer) 那里所得到的，我只能希望读者把这份爱继续传递下去。

——Nicholas

目 录

第 1 章 Bower 1

- 1.1 准备工作 1
- 1.2 配置 Bower 2
- 1.3 清单文件 (Manifest) 2
- 1.4 查找、添加和删除 Bower 包 3
 - 1.4.1 查找包 3
 - 1.4.2 添加包 3
 - 1.4.3 删除包 5
- 1.5 语义化版本控制 5
- 1.6 维护依赖链 6
- 1.7 创建 Bower 包 7
 - 1.7.1 选择有效的包名 7
 - 1.7.2 在 Git 标签中使用语义化版本号 (Semver) 7
 - 1.7.3 将软件包发布到注册中心 7
- 1.8 小结 8

第 2 章 Grunt 9

- 2.1 安装 Grunt 10
- 2.2 Grunt 是如何工作的 10
 - 2.2.1 Gruntfile.js 10
 - 2.2.2 任务 (Tasks) 11
 - 2.2.3 插件 (Plugins) 11
 - 2.2.4 配置 12
- 2.3 将 Grunt 添加到项目中 12
- 2.4 处理任务 14
 - 2.4.1 配置管理 14
 - 2.4.2 任务描述 15
 - 2.4.3 异步任务 15
 - 2.4.4 任务依赖 16
 - 2.4.5 多任务 16

- 2.4.6 多任务选项 17

- 2.4.7 模板配置 18

- 2.4.8 命令行选项 19

- 2.4.9 提供反馈 19

- 2.4.10 错误处理 20

- 2.5 操作文件系统 20

- 2.5.1 源-目标映射 20

- 2.5.2 监视文件变化 22

- 2.6 创建 Grunt 插件 25

- 2.6.1 开始 25

- 2.6.2 创建任务 26

- 2.6.3 将任务发布到 npm 28

- 2.7 小结 28

- 2.8 相关资源 29

第 3 章 Yeoman 30

- 3.1 安装 Yeoman 30

- 3.2 创建第一个项目 30

- 3.3 创建你的第一个脚手架 34

- 3.3.1 Yeoman 脚手架是一个 Node 模块 34

- 3.3.2 子脚手架 35

- 3.3.3 定义二级命令 39

- 3.3.4 可组合性 41

- 3.4 小结 41

- 3.5 相关资源 42

第 4 章 PM2 43

- 4.1 安装 43

- 4.2 与进程一起工作 43

- 4.2.1 从错误中恢复 46

- 4.2.2 监控文件变化 47

4.3	监控日志	48
4.4	监控资源占用	49
4.4.1	监控本地资源	49
4.4.2	监控远程资源	49
4.5	进程的高级管理	52
4.6	多核处理器的负载均衡	55
4.7	小结	59
4.8	相关资源	59

第 5 章 RequireJS

5.1	运行示例	61
5.2	使用 RequireJS	61
5.2.1	安装	62
5.2.2	配置	62
5.2.3	应用模块和依赖	64
5.2.4	路径和别名	66
5.2.5	Shims	69
5.2.6	加载器插件	73
5.2.7	缓存清除	78
5.3	RequireJS 优化	80
5.3.1	配置 r.js	80
5.3.2	运行 r.js 命令	81
5.4	小结	82

第 6 章 Browserify

6.1	AMD API 与 CommonJS 对比	84
6.2	安装 Browserify	85
6.3	创建你的第一个 Bundle	85
6.4	可视化依赖树	87
6.5	发生变化时重新打包文件	88
6.5.1	通过 Grunt 监听文件变化	88
6.5.2	通过 Watchify 监听文件 变化	88
6.6	使用多个打包文件	90
6.7	Node 方式	92
6.7.1	模块解析方案和 NODE_PATH 环境变量	93
6.7.2	依赖管理	95
6.8	定义浏览器指定模块	96

6.9	用 Transforms 扩展 Browserify	97
6.9.1	brfs	97
6.9.2	folderify	98
6.9.3	bulkify	98
6.9.4	Browserify-Shim	99
6.10	小结	100
6.11	相关资源	100

第 7 章 Knockout

7.1	View、Model 与 View Model	102
7.1.1	菜谱列表	103
7.1.2	菜谱详情	106
7.2	将视图绑定到 DOM	108
7.3	视图模型与表单	109
7.3.1	切换到“编辑”模式	109
7.3.2	更改菜谱的标题	112
7.3.3	更改菜谱的份量与 烹饪时间	112
7.3.4	添加与删除食材	114
7.3.5	操作步骤	118
7.3.6	引文	119
7.4	自定义组件	120
7.4.1	input-list 组件的视图模型	120
7.4.2	input-list 模板	121
7.4.3	注册 input-list 组件	123
7.5	Subscribable: 简单的消息传递	124
7.6	小结	126
7.7	相关资源	127

第 8 章 AngularJS

8.1	声明式 Web 编程	128
8.1.1	命令式编程	128
8.1.2	声明式编程	129
8.2	模块: 构建松散耦合程序的基石	130
8.3	指令 (Directive): DOM 的抽 象层	132
8.4	加入逻辑	134
8.4.1	作用域与原型继承	134
8.4.2	用控制器操作作用域	135

8.5 通过服务与依赖注入实现松散耦合	138	10.2 安装	203
8.5.1 依赖注入 (Dependency Injection)	138	10.3 Mach Web 服务	203
8.5.2 简单的控制器与复杂的服务	139	10.3.1 HTTP 路由	205
8.6 创建路由	142	10.3.2 建立连接	210
8.6.1 路由参数	143	10.3.3 公共的中间件	212
8.6.2 路由的 Resolve	144	10.3.4 路由重写	226
8.7 创建复杂表单	145	10.3.5 主机映射	228
8.7.1 表单验证	146	10.3.6 自定义中间件	232
8.7.2 条件逻辑	150	10.4 Mach HTTP 客户端	234
8.7.3 列表	151	10.5 Mach HTTP 代理	236
8.8 小结	153	10.6 小结	239
8.9 相关资源	154	第 11 章 Mongoose	240
第 9 章 Kraken	155	11.1 MongoDB 的基本概念	240
9.1 环境感知的配置	156	11.2 Mongoose 的一个简单示例	243
9.2 注册基于配置的中间件	162	11.2.1 针对 JSON 数据创建一个 Mongoose 模式	243
9.3 结构化路由注册	165	11.2.2 使用 Mongoose 导入数据	244
9.3.1 索引配置	165	11.2.3 通过 Mongoose 查询数据	247
9.3.2 目录配置	166	11.3 使用模式 (Schemas)	248
9.3.3 路由配置	167	11.3.1 数据类型	248
9.4 Dust 模板	169	11.3.2 嵌套模式	250
9.4.1 上下文及引用	169	11.3.3 默认属性值	250
9.4.2 片段	171	11.3.4 必要属性	251
9.4.3 迭代	172	11.3.5 第二索引	251
9.4.4 条件句	173	11.3.6 模式校验	252
9.4.5 局部模板	173	11.3.7 模式引用	255
9.4.6 块	174	11.3.8 模式中间件	258
9.4.7 过滤器	175	11.4 使用模型和文档	259
9.4.8 上下文辅助器	176	11.4.1 文档实例方法	262
9.4.9 Dust 辅助器	182	11.4.2 文档虚拟属性	263
9.4.10 使用 Kraken	186	11.4.3 静态模型方法	265
9.5 小结	200	11.5 使用查询	266
9.6 相关资源	200	11.5.1 Model.find()	266
第 10 章 Mach	202	11.5.2 使用查询运算符查找文档	272
10.1 章节例子	202	11.6 小结	278

第 12 章 Knex 和 Bookshelf	279
12.1 Knex	279
12.1.1 安装命令行工具	280
12.1.2 把 Knex 添加到你的项目	280
12.1.3 配置 Knex	280
12.1.4 SQL 查询构建器	281
12.1.5 迁移脚本	287
12.1.6 种子脚本	291
12.2 Bookshelf	291
12.2.1 什么是对象映射关系	292
12.2.2 创建 Bookshelf 模型	292
12.2.3 关系	299
12.3 小结	306
12.4 相关资源	307
第 13 章 Faye	308
13.1 HTTP、Bayeux 和 WebSocket	308
13.1.1 WebSocket	310
13.1.2 Bayeux 协议	310
13.2 开始使用 Faye	312
13.3 发布/订阅消息系统	313
13.4 小结	318
13.5 相关资源	318
第 14 章 Q	319
14.1 时间就是一切	319
14.2 Promise 对比回调函数	322
14.3 Q 的 Promise	324
14.3.1 Deferreds 和 Promises	324
14.3.2 值和错误	328
14.3.3 报告进度	333
14.3.4 终点	336
14.4 控制流	338
14.4.1 顺序流	338
14.4.2 平行流	339
14.4.3 管道流	341
14.5 小结	342
14.6 相关资源	343

第 15 章 Async.js	344
15.1 顺序流	345
15.2 并行流	346
15.3 管线流	348
15.4 循环流	352
15.4.1 为真则循环执行	352
15.4.2 为假则循环执行	354
15.4.3 循环重试	355
15.4.4 无限循环	357
15.5 批处理流	358
15.5.1 异步队列	358
15.5.2 异步负载	359
15.6 小结	361
第 16 章 Underscore 和 Lodash	362
16.1 安装及用法	363
16.2 聚合和索引	364
16.2.1 countBy()	364
16.2.2 groupBy()	365
16.2.3 indexBy()	366
16.3 选择	367
16.3.1 从集合中选择数据	367
16.3.2 从对象中选择数据	369
16.4 链式调用	373
16.5 函数计时	375
16.5.1 defer()	375
16.5.2 debounce()	377
16.5.3 throttle()	378
16.6 模板	380
16.6.1 模板内的循环及其他 JavaScript 代码	381
16.6.2 书写不加鳄鱼标记的代码	382
16.6.3 从模板中获取数据对象	383
16.6.4 默认模板数据	384
16.7 小结	385
16.8 相关资源	386

Bower

九层之台，起于累土。

—— 文森特·梵高

包管理 (Package Management)，又作**依赖关系管理 (Dependency Management)**，并不是什么新奇的概念。此类工具为开发者提供了一种机制，以管理软件项目所依赖的各种第三方库。一些得到广泛应用的例子有：

- **npm**: Node.js 的包管理工具；
- **Composer**: 一种 PHP 依赖关系管理工具；
- **pip**: PyPA 的推荐工具，用于安装 Python 包；
- **NuGet**: 包括 .NET 在内的微软开发平台的包管理工具。

尽管包管理并不是新概念，但是将其广泛应用于前端资源的管理却是最近才有的事情。这些资源包括 JavaScript 库、样式表、字体、图标 (icon) 以及图像等，它们是现代网络应用的基本构件。随着现代网络应用的构建基础越来越复杂，对包管理工具的需求日益凸显。那些曾经以某些万金油式的第三方库 (如 jQuery) 为基础开发出的网络应用，也开始逐渐转向一些功能专一且体积小巧的库。这样做使得软件模块更加小巧，从而易于测试，同时应用的灵活性也得到了加强，可以方便地通过第三方库进行扩展，并在必要的时候进行替换。

本章旨在让你快速上手 Bower，这是一款源自 Twitter 开源项目的前端包管理工具。本章涵盖的主题有：

- Bower 的安装和配置；
- 在项目中添加 Bower；
- 查找、添加和删除包；
- 语义化版本控制；
- 管理依赖链；
- 创建 Bower 包。

1.1 准备工作

用户与 Bower 之间的所有交互都通过命令行工具来完成，该工具可以通过 npm 安装。如果你还没有安装 Bower，那么请在继续阅读之前，按照清单 1-1 进行安装。

清单 1-1 通过 npm 安装 Bower 命令行工具

```
$ npm install -g bower
$ bower --version
1.3.12
```

■ **注意** Node 的包管理工具 (npm) 允许用户将软件包安装到局部或全局两个环境之一。在本例中, 我们将 Bower 安装到了全局环境。通常, 全局环境用于安装各种命令行工具。

1.2 配置 Bower

Bower 采用基于项目的配置, 也就是说, 每个项目都可以通过一个 (可选的) JSON 文件进行配置。该文件位于项目的根目录下, 文件名为 `.bowerrc`。简而言之, 我们只看一下该文件中变动最频繁的设置 (见清单 1-2)。

清单 1-2 本章示例项目中的 `.bowerrc` 文件

```
// example-bootstrap/.bowerrc

{
  "directory": "../public/bower_components"
}
```

默认情况下, Bower 会把项目的所有依赖项存储在 `bower_components` 文件夹下。如果想要改变依赖项的存储位置, 只需修改 `directory` 配置项即可。

1.3 清单文件 (Manifest)

Bower 为开发者查找、添加、升级以及删除第三方库提供了唯一入口。这些操作被执行后, Bower 会用最新的项目依赖项列表更新一个被称作“配置清单”的 JSON 文件。本章示例项目中的配置清单如清单 1-3 所示。在此示例项目中, Bower 维护着一个唯一的依赖项, 即 Bootstrap 的 CSS 框架。

清单 1-3 本章示例项目的 Bower 清单文件

```
// example-bootstrap/bower.json

{
  "name": "example-bootstrap",
  "version": "1.0.0",
  "homepage": "https://github.com/username/project",
  "authors": [
    "John Doe <john.doe@gmail.com>"
  ],
  "dependencies": {
    "bootstrap": "3.2.0"
  }
}
```

如果不小心把 `/public/bower_components` 文件夹删除, 以致弄丢项目所有的依赖项的话, 只需一条命令 (如下所示) 就可以将项目恢复至原始状态。Bower 会将项目当前的文件结构与清单文

件进行比对，确定缺失了哪些依赖项，最终重建项目模式。

```
$ bower install
```

这种特性使得我们可以不对 `/public/bower_components` 文件夹进行版本管理。提交代码时只提交 Bower 清单文件，而不提交依赖项本身，项目的源代码模式得以保持整洁，仅包含项目本身所有的文件。

■ **注意** 对于是否将项目中的依赖项移出版本管理存在分歧。一方面，这么做可以令项目仓库更加整洁；另一方面，这也为由网络连接引发的问题（比如不能访问 Bower 注册中心或者 GitHub 等）埋下了隐患。普遍共识是，如果你的项目是可部署的（亦即一个完整的应用，而非软件模块），那么最好将依赖项一起提交，否则最好将项目用到的依赖项移出版本管理。

创建新的清单文件

当在项目中初次使用 Bower 的时候，最好像下面一样，让 Bower 创建一个新的清单文件。此后，再根据需要进行修改。

```
$ bower init
```

1.4 查找、添加和删除 Bower 包

Bower 的命令行工具提供了很多有用的命令，用于查找、安装或者删除包。我们来看一下这些命令是如何简化项目外部依赖项的管理工作的。

1.4.1 查找包

Bower 改善开发流程的一个主要途径就是为第三方库提供集中式的注册中心。如清单 1-4 所示，要在 Bower 的注册中心里进行搜索的话，只需将 `search` 参数传给 Bower 即可，`search` 后面跟着要查找的关键词。下面列出的仅为查询结果的一个小片段。

清单 1-4 在 Bower 中查找 jQuery

```
$ bower search jquery
```

Search results:

```
jquery git://github.com/jquery/jquery.git
jquery-ui git://github.com/components/jqueryui
jquery.cookie git://github.com/carhartl/jquery-cookie.git
jquery-placeholder git://github.com/mathiasbynens/jquery-placeholder.git
```

1.4.2 添加包

每一条搜索结果都由包的注册名与其 GitHub 仓库的 URL 构成，由此 URL 可以直接访问该包的所有信息。找到所需的包后，即可将其添加到项目中，如清单 1-5 所示。

清单 1-5 将 jQuery 添加到项目中

```
$ bower install jquery --save
bower jquery#*          cached git://github.com/jquery/jquery.git#2.1.3
bower jquery#*          validate 2.1.3 against git://github.com/jquery/jquery.git#*
bower jquery#>= 1.9.1    cached git://github.com/jquery/jquery.git#2.1.3
bower jquery#>= 1.9.1    validate 2.1.3 against git://github.com/jquery/jquery.git#>= 1.9.1
bower jquery#>= 1.9.1    cached git://github.com/jquery/jquery.git#2.1.3
bower jquery#>= 1.9.1    validate 2.1.3 against git://github.com/jquery/jquery.git#>= 1.9.1
bower jquery#>= 1.9.1    install jquery#2.1.3
```

```
jquery#2.1.3 public/bower_components/jquery
```

■ **注意** Bower 并没有在其注册中心中托管与包相关的任何文件，这个任务是由 GitHub 完成的。尽管理论上讲，可以将软件包托管到任何 URL，但是大多数公有域的包还是在 GitHub 上。

注意清单 1-5 中跟在 Bower 的 `install` 命令后面的 `--save` 选项。默认情况下，`install` 命令仅将包添加到项目中，而不更新项目清单。`--save` 选项指示 Bower 将该包永久保存在项目的依赖列表里面。

清单 1-6 中显示的是本章示例项目中的 HTML 代码。利用 Bower 将 jQuery 添加到项目中后，即可像加载其他库一样，通过 `script` 标签将其加载到页面中。

清单 1-6 来自示例项目的 HTML，该文件引用了刚刚添加的 jQuery 库

```
// example-jquery/public/index.html
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bower Example</title>
  </head>
  <body>
    <div id="container"></div>
    <script src="/bower_components/jquery/dist/jquery.min.js"></script>
    <script>
      $(document).ready(function() {
        $('#container').html('<p>Hello, world!</p>');
      });
    </script>
  </body>
</html>
```

开发版依赖项（Development Dependencies）

默认情况下，Bower 安装的所有包都是“产品”版的依赖项（Production Dependencies）。传入 `--save-dev` 选项可以改变这一行为，从而安装的所有包都将被标记为“开发”版。开发版软件包通常仅用于开发目的，而不面向项目的最终用户。

一旦准备好将应用部署到生产环境中，就可以按照下面的方法，让 Bower 将项目依赖项切换到产品版，从而使项目构建更加紧凑，不含与最终用户无关的文件。

```
$ bower install --production
```


1.4.3 删除包

删除包的方法非常简单。与前面一样，我们通过添加`--save`选项来更新 Bower 的清单文件，以反映对项目所做的更改。

```
$ bower uninstall jquery --save
```

1.5 语义化版本控制

如果安装了 jQuery（见清单 1-5），那么请查看项目的 Bower 清单。你所看到的应当与清单 1-7 相仿。

清单 1-7 语义化版本号

```
"dependencies": {  
  "jquery": "~2.1.3"  
}
```

清单 1-7 中的版本号 2.1.3（请暂时忽略~）就是所谓的语义化版本号（Semver, Semantic Version）。语义化版本控制能够帮助开发者按照通用格式为其项目指定版本号。该格式如下所示。

Version X.Y.Z(Major.Minor.Patch)

语义化版本格式要求开发者创建定义明晰的 API（通过文档或者自文档化的代码），从而为用户使用软件库提供唯一的切入点。新启动的项目一般从版本 0.0.0 开始，然后随着新版本的推出，版本号逐渐增加。通常认为版本号在 1.0.0 以下的项目正处于紧密开发中，此时允许在不变更主版本号（Major）的情况下，对其 API 进行重大改变。而版本号 1.0.0 及以上的项目，在更改版本号时需要遵守以下规则。

- 当更新导致用户使用项目 API 的方式发生重大变化的时候，项目的主版本号应当发生改变。
- 当以向后兼容的方式添加新特性的时候（也就是说，现有 API 不会失效），项目的次版本号应当发生改变。
- 当以向后兼容的方式修正 bug 的时候，项目的修订号（Patch Version Number）应当发生改变。

这些规则可以让开发者了解任意两个版本之间的变化程度。随着 Bower 清单的增长以及项目所需依赖项的增多，这些信息的作用会逐渐显现出来。

■ **注意** 清单 1-7 中的“~”号表示无论何时运行 `install` 命令，都允许以“相对接近”2.1.3 的版本对 jQuery 进行升级。如果“相对接近”和“自动安装”这样的字眼放在一起使用让你觉得浑身不舒服的话，那就对了！最佳实践建议，在 Bower 中引用依赖项的时候，应当避免使用“~X.Y.Z”这样的格式。相反，你最好明确指定要在项目中引入的依赖项的版本。当有更新发布时，你可以手动检查，并决定是否以及何时进行升级。本章随后的示例将会遵从此建议。

1.6 维护依赖链

Bower 给开发者带来的一个主要好处，就是可以非常方便地对整个项目的依赖链以一种相对受控的方式进行集中升级。为此，我们先来看一下本章示例项目所包含的依赖项列表（见清单 1-8）。

清单 1-8 安装并列出示例项目所需的 Bower 包

```
$ bower install
bower bootstrap#3.2.0   cached git://github.com/twbs/bootstrap.git#3.2.0
bower bootstrap#3.2.0   validate 3.2.0 against git://github.com/twbs/bootstrap.git#3.2.0
bower jquery#>= 1.9.0    cached git://github.com/jquery/jquery.git#2.1.3
bower jquery#>= 1.9.0    validate 2.1.3 against git://github.com/jquery/jquery.git#>= 1.9.0
bower bootstrap#3.2.0    install bootstrap#3.2.0
bower jquery#>= 1.9.0    install jquery#2.1.3

bootstrap#3.2.0 public/bower_components/bootstrap
└─ jquery#2.1.3

jquery#2.1.3 public/bower_components/jquery

$ bower list
bower check-new Checking for new versions of the project dependencies...
example-bootstrap#1.0.0 /opt/example-bootstrap
└─ bootstrap#3.2.0 (latest is 3.3.2)
   └─ jquery#2.1.3
```

多亏 Bower 为我们提供了这样简单的示意图，来描述项目所需的依赖项以及它们之间的关系。可以看到，我们的项目依赖 Bootstrap，而 Bootstrap 又依赖 jQuery。此外，Bower 还为我们打印出了每个组件当前安装的版本。

注意 许多第三方库并不是自包含（self-contained，亦即独立自足的），Bootstrap（它有依赖项 jQuery）就是一个例子。当添加这样的包的时候，Bower 可以非常智能地识别出额外的依赖项，并在缺失时主动将其添加到项目中。但需要注意的是，与那些复杂的包管理工具（例如 npm）不同，Bower 使用平坦的文件夹结构（Flat Folder Structure，指文件夹模式没有深层次的嵌套）来存储所有的软件包。这就意味着如果不小心的话，你可能会遇到版本冲突的问题。

在清单 1-8 中，Bower 提示 Bootstrap 有一个版本号为 3.3.2 的可用更新（当前项目中的 Bootstrap 为 3.2.0 版）。要升级此依赖项，只需要在项目清单文件中引用新版本，然后重新运行 `install` 命令即可，如清单 1-9 所示。

清单 1-9 在更新项目所依赖的 Bootstrap 之后，重新安装 Bower 包

```
$ bower install
bower bootstrap#3.3.2   cached git://github.com/twbs/bootstrap.git#3.3.2
bower bootstrap#3.3.2   validate 3.3.2 against git://github.com/twbs/bootstrap.git#3.3.2
bower bootstrap#3.3.2   install bootstrap#3.3.2

bootstrap#3.3.2 public/bower_components/bootstrap
└─ jquery#2.1.3
```

1.7 创建 Bower 包

截至目前，我们关注的焦点都在如何将 Bower 集成到项目中。我们先在项目中对 Bower 进行了初始化，然后探究了查找、添加以及移除软件包的方法。但是总有那么一天，你会希望将自己的软件包分享给其他人。要这么做的话，就必须遵守一些简单的规则，让我们从选择有效的包名开始。

1.7.1 选择有效的包名

你需要为自己的包选定一个名字，该名字必须在整个 Bower 开放注册中心（public registry）唯一。通过 Bower 的 `search` 命令来检查你想要的名字是否可用。其他需要遵守的规则有：

- 包名应当使用 slug 格式，例如 `my-unique-project`。
- 包名中的所有字母应为小写。
- 只允许出现字母、句点以及连字符（dash）。
- 以字母起始并结束。
- 不允许出现连续的句点和连字符。
- 选好名字后，更新相应的 `bower.json` 文件的内容。

1.7.2 在 Git 标签中使用语义化版本号（Semver）

本章前面的部分对语义化版本控制（一种为软件项目指定有意义的版本号的通用规范）的概念进行了介绍。请务必遵守此规范，因为它使用户得以跟踪及整合软件包以后的改变。

如果你要共享的软件包刚刚起步，`0.0.0` 会是比较合适的版本号。随着更新的提交以及新版本的发布，你可以根据改变程度相应地增加版本号。当确定该项目已经到达第一个“稳定的”里程碑的时候，再把版本号更新为 `1.0.0` 来反映此状态。

软件项目的每个版本号都应该在 GitHub 上有相应的标签（tag）。用户正是通过这种 GitHub 标签与包版本号之间的关系，在项目中引用特定版本的包的。

假设代码已经提交到 GitHub 上，那么接着可以按清单 1-10 所示的方法来创建第一个 GitHub 标签。

清单 1-10 创建第一个使用语义化版本号的 Git 标签

```
$ git tag -a 0.0.1 -m "First release."  
$ git push origin 0.0.1
```

1.7.3 将软件包发布到注册中心

软件包的名字已经选好，也指定了版本号（以及相应的 GitHub 标签），现在是时候将软件包发布到 Bower 的注册中心了。

```
$ bower register my-package-name https://github.com/username/my-package-name.git
```

注意 请记住，Bower 的设计初衷是作为库和组件的集中式注册中心，供开发者在项目中使用，而非一种应用分发机制。

1.8 小结

Bower 是一个简单的命令行工具，可以用来简化前端资源管理过程中的一些冗杂的工作。与其他平台上的知名包管理工具（如 **npm**）不同，**Bower** 的设计初衷并不是解决某种平台或者语言的特定需求；相反，它支持的是一般意义上的包管理。创建 **Bower** 的开发者有意创造了一个能够管理众多种类的前端资源的简单工具，不仅仅是代码，还包括样式表、字体、图像以及其他未预见的依赖项。

与普通小型网络应用打交道的开发人员可能会觉得 Bower 带来的好处价值不大。然而，小网络应用都有快速迭代为复杂网络应用的倾向。往往到那个时候，开发者才会感激 Bower 带来的益处。

无论你的项目多么复杂（或者多么简单），我们都建议你尽快将 **Bower** 集成到工作流中，因为我们已经吃过苦头（来自项目本身的）了。在项目结构方面做的工作太少，就会产生背负技术债务的风险，并且债务会不断增加，总有一天会让你付出代价。在这些令人为难的选择之间达成微妙平衡，既是科学，也是艺术。这一过程也从来没有得到完全的认识，随着软件工具的更新换代。我们必须不断地做出调整。

Grunt

我是一个懒人。但正是懒人发明了轮子和自行车，因为他们既不愿意走路，也不愿意负重前行。
——莱赫·瓦文萨，波兰前总统

拉里·沃尔（Larry Wall，著名的 Perl 语言创始人）在他的《Programming Perl》一书中提到，所有成功的程序员都有三个重要的品质：懒惰、急躁以及狂妄。乍一看，这些都是相当糟糕的品质，但是只是深挖一下，你就会发现其言外之意。

懒惰：懒惰的程序员讨厌重复自己。他们通常会花费大量的时间去创造有用的工具，代替自己完成重复性的工作。他们往往还会为这些工具编写详尽的文档，以免之后可能因此而遇到麻烦。

急躁：没耐心的程序员通常会对他们的工具抱以高度的期望。这种期望使得他们编写的软件不仅能够满足用户的需求，而且会对用户的需求做出合理的预期。

狂妄：优秀的程序员以他们的工作为傲。正是这份骄傲驱使他们创造出令人赞叹的软件，这是我们都应该为之奋斗的目标。

本章以 Grunt 为例，阐释三个品质中的第一个——懒惰。Grunt 是流行的 JavaScript 构建工具，它所提供的工具包能够帮助开发者自动执行开发过程中遇到的重复性构建任务，从而助长开发者的惰性，例如：

- 脚本以及样式表的编译和压缩（minification）
- 测试
- 静态检查（linting）
- 数据库迁移
- 部署

换句话说，Grunt 帮助开发者更加聪明地工作，而不是让他们变得庸庸碌碌。如果你觉得这个主意不错的话，那就继续阅读吧。读完本章之时，你会发现自己已经踏上了通往 Grunt 大师的道路。在本章中，你将会学到如下内容。

- 创建可配置的任务以自动化执行软件开发中的重复工作，这些重复工作几乎存在于所有的工程中。
- 通过 Grunt 提供的简单且强大的抽象能力与文件系统进行交互。
- 发布 Grunt 插件来造福其他的开发者，而且可以让他们参与改进。
- 充分利用 Grunt 业已存在的插件库。这些库由技术社区提供支持。本书撰写之时，已经有超过 4 400 个示例库存在。

2.1 安装 Grunt

继续之前，请确保 Grunt 的命令行工具已经安装妥当。Grunt 的命令行工具以 npm 包的形式发行，其安装过程如清单 2-1 所示。

清单 2-1 通过 npm 安装 Grunt 命令行工具

```
$ npm install -g grunt-cli
$ grunt --version
grunt-cli v0.1.13
```

2.2 Grunt 是如何工作的

Grunt 为开发者提供了一个工具包，用于创建命令行程序来执行项目构建过程中的重复性任务，如压缩 JavaScript 代码、编译 Sass 样式表等。不过，Grunt 的能力并不限于创建简单的任务（通常这些任务不会被分享或者复用），以解决特定工程遇到的特定需求，其真正的力量源于其将任务打包为可复用的插件的能力。这些插件可以被发布、分享、使用以及由其他人进行改进。本书写作之时已经有超过 4 400 个这样的插件。

Grunt 的运转依赖于四个核心组件，接下来逐一论述。

2.2.1 Gruntfile.js

在 Grunt 中处于核心地位的是 *Gruntfile*——一个位于工程根目录下的名为 *Gruntfile.js*（见清单 2-2）的 Node 模块。正是这个文件使得我们可以加载 Grunt 插件，创建自定义任务，并根据项目需求对它们进行配置。Grunt 每次运行时的首要任务都是接受该模块发出的指令。

清单 2-2 Gruntfile 示例

```
// example-starter/Gruntfile.js

module.exports = function(grunt) {
```

```
  /**
   * 配置即将用到的任务和插件
   */
  grunt.initConfig({
    /* Grunt 的 file API 为开发者提供了与文件系统进行交互所必需的抽象。稍后，我们将在
     本章对此进行深入了解。*/
    'pkg': grunt.file.readJSON('package.json'),
    'uglify': {
      'development': {
        'files': {
          'build/app.min.js': ['src/app.js', 'src/lib.js']
        }
      }
    }
  });
  /**
```

* Grunt 插件以 Node 包的形式存在，并由 npm 发布。这里，我们加载的是 *grunt-contrib-uglify* 插件。

```

* 该插件包含的任务可以对项目源代码进行合并与压缩，以备发布之用。
*/
grunt.loadNpmTasks('grunt-contrib-uglify');

/**
 * 这里，我们创建了一个名为 default 的任务，其仅有的功能就是调用 uglify 任务。换句话说，该任务
 * 实际上是 uglify 任务的别名。名为 default 的任务指明了在命令行中不带参数调用 Grunt 时应当执
 * 行的动作。在本例中，我们的 default 任务仅仅调用了单独的任务。不过（依次）调用多个任务
 * 其实同样简单，只要在传入的数组中添加多个条目即可。
 */
grunt.registerTask('default', ['uglify']);

/**
 * 这里，我们创建了一个自定义任务，利用 Grunt 内置的用户反馈（user feedback）方法，向控制台
 * 输出一条消息（后面还有一个换行符）。稍后，我们将在本章对此做深入了解。
 */
grunt.registerTask('hello-world', function() {
  grunt.log.writeln('Hello, world.');
```

```
});
```

2.2.2 任务 (Tasks)

作为 Grunt 的基本构建模块，任务实际上只是由 Grunt 的 `registerTask()` 方法注册的具名函数。清单 2-2 所示的 `hello-world` 任务将向控制台输出一条消息。在命令行中调用该任务的结果，如清单 2-3 所示。

清单 2-3 运行清单 2-2 中所示的 `hello-world` 任务

```

$ grunt hello-world
Running "hello-world" task
Hello, world.

Done, without errors.
```

如清单 2-4 所示，多个 Grunt 任务也可以由单条命令调用执行。每个任务都将按照参数的传入顺序依次执行。

清单 2-4 顺序运行多个任务

```

$ grunt hello-world uglify
Running "hello-world" task
Hello, world.

Running "uglify:development" (uglify) task
>> 1 file created.

Done, without errors.
```

我们刚看到的 `hello-world` 任务是简单独立型 Grunt 任务的代表。这样的任务可以用于实现一些简单的功能，以解决特定项目的需求。通常我们不会考虑其复用或者分享的问题。但是多数时候，你会发现我们实际用到的都不是这样的独立型任务，而是那些已经打包为 Grunt 插件并发布到 npm 的任务。以插件的形式发布更便于别人使用或者参与改进。

2.2.3 插件 (Plugins)

Grunt 插件是一系列能够用于不同项目的可配置任务（以 npm 包的形式发布）的集合。现存的 Grunt

插件数以千计，可谓洋洋大观。清单 2-2 中的 Grunt 方法 `loadNpmTasks()` 用以加载名为 `grunt-contrib-uglify` 的 Node 模块。该模块可以将工程中的 JavaScript 代码合并为单个压缩 (minified) 文件，以适应发布需求。

■ **注意** <http://gruntjs.com/plugins> 中列出了所有可用的 Grunt 插件。名字带有 `contrib-` 前缀的插件由 Grunt 官方的开发者进行维护。

2.2.4 配置

Grunt 以强调“配置优先” (configuration over code) 而著称：任务和插件的功能均可通过配置文件进行定制，以适应不同工程的需求。正是这种代码与配置分离的特性，使开发者能够创造出容易被复用的插件。本章稍后将介绍配置 Grunt 插件和任务的各种不同的方法。

2.3 将 Grunt 添加到项目中

在本章前面，为了添加 Grunt 命令行工具，我们将 npm 包 `grunt-cli` 作为全局模块进行了安装。现在我们应该已经可以在命令行中使用 Grunt 命令，但是对于每个要使用 Grunt 的工程，仍然需要为其配置 Grunt 本地依赖。为此，只需在工程根目录下运行以下命令即可。本例假设 npm 已经针对示例所用项目进行了初始化，`package.json` 文件也已经存在。

```
$ npm install grunt --save-dev
```

现在，我们项目的 `package.json` 文件应该已经包含与清单 2-5 中的示例代码相仿的 Grunt 条目。

清单 2-5 更新后的 `package.json` 文件

```
// example-tasks/package.json
```

```
{
  "name": "example-tasks",
  "version": "1.0.0",
  "devDependencies": {
    "grunt": "0.4.5"
  }
}
```

将 Grunt 与项目进行整合的最后一个步骤是创建 `Gruntfile` (见清单 2-6)，并保存在工程根目录下。我们的 `Gruntfile` 只调用了函数 `loadTasks()`，其作用将在接下来的小节进行讨论。

清单 2-6 我们的 `Gruntfile`

```
// example-tasks/Gruntfile.js
```

```
module.exports = function(grunt) {
  grunt.loadTasks('tasks');
};
```

保持合理的 Grunt 模式

我们希望你读完本章之后发现，Grunt 的确是一件处理日常工作流程中遇到的乏味的重复性工作的利

器。但是即便如此，还是不得不承认我们对 Grunt 的第一反应绝对算不上积极。事实上，刚开始我们对它兴趣不大。为了了解其中的原因，我们看一下放在 Grunt 官方文档显著位置的 Gruntfile 示例（见清单 2-7）。

清单 2-7 Grunt 官方文档提供的 Gruntfile 示例

```
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      options: {
        separator: ';'
      },
      dist: {
        src: ['src/**/*.js'],
        dest: 'dist/<%= pkg.name %>.js'
      }
    },
    uglify: {
      options: {
        banner: '/*! <%= grunt.template.today("dd-mm-yyyy") %> */\n'
      },
      dist: {
        files: {
          'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
        }
      }
    },
    qunit: {
      files: ['test/**/*.html']
    },
    jshint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        // options here to override JSHint defaults
        globals: {
          jquery: true,
          console: true,
          module: true,
          document: true
        }
      }
    },
    watch: {
      files: ['<%= jshint.files %>'],
      tasks: ['jshint', 'qunit']
    }
  });

  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-qunit');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-concat');

  grunt.registerTask('test', ['jshint', 'qunit']);

  grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);
};
```

清单 2-7 中略显臃肿的 Gruntfile 还是为相对简单的项目准备的。对于规模更大的项目，Gruntfile 会像

吹气球一样膨胀好几倍，最终结果就是一大堆根本无法阅读又难以维护的东西。有经验的开发者绝不会在写代码的时候，将不相关的功能模块揉在一起。否则，我们为什么还要在任务运行器上搞点儿花样出来呢？

正如清单 2-6 中的代码所示，使 Grunt 模式保持合理的秘密就藏在 Grunt 的 `loadTasks()` 方法上。该例中 `task` 参数指向 `task` 文件夹，其路径以工程的 `Gruntfile` 为参考位置。`loadTask()` 方法一被调用，Grunt 就会加载并执行在该文件夹下发现的所有 Node 模块，并且每次都把 Grunt 对象的引用作为参数传进去。这样的行为使我们可以将项目的 Grunt 配置划分为一系列相互独立的模块，每个模块负责一个单独的任务或插件的加载及配置工作。清单 2-8 展示了其中一个小型模块，模块中定义的任务可以通过在命令行中执行 `grunt uglify` 来运行。

清单 2-8 task 文件夹中的示例模块 (`uglify.js`) *

```
// example-tasks/tasks/uglify.js
module.exports = function(grunt) {
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.config('uglify', {
        'options': {
            'banner': '/*! <%= grunt.template.today("dd-mm-yyyy") %> */\n'
        },
        'dist': {
            'files': {
                'dist/app.min.js': ['src/index.js']
            }
        }
    });
};
```

2.4 处理任务

如前所述，任务在 Grunt 中扮演着基石一般的角色。在 Grunt 的世界里，任务乃万物之源。如你所见，Grunt 插件只是打包为 Node 模块并通过 npm 发布的一个或多个任务而已。我们已经见过一些创建基本 Grunt 任务的例子，接下来看看还有哪些特性能够让我们充分利用。

2.4.1 配置管理

对于配置来说，Grunt 的 `config()` 方法既是“getter”，也是“setter”。在清单 2-9 中，我们可以看到一个基本的 Grunt 任务是如何通过此方法来存取配置的。

清单 2-9 管理一个基本 Grunt 任务的配置

```
module.exports = function(grunt) {
    grunt.config('basic-task', {
        'message': 'Hello, world.'
    });
    grunt.registerTask('basic-task', function() {
        grunt.log.writeln(grunt.config('basic-task.message'));
    });
};
```

■ **注意** 清单 2-9 中“点记法” (dot notation) 是用来访问多级嵌套的配置参数的。采用同样的方式, 点记法还可以用于设置多级嵌套的配置参数值。而且无论在哪里遇到不存在的配置路径, Grunt 都会为其创建一个新的空对象, 而不抛出异常。

2.4.2 任务描述

随着时间的推移, 项目复杂性会逐渐增加, 因此需要不断添加新的 Grunt 任务。但是随着任务的增多, 任务的可用性、用途以及调用方法等越来越难以追踪。幸运的是, Grunt 已经为我们提供了一条解决此问题的途径。如清单 2-10 所示, 我们可以为任务设定相应的描述。

清单 2-10 为 Grunt 任务设置描述

```
// example-task-description/Gruntfile.js
```

```
module.exports = function(grunt) {

  grunt.config('basic-task', {
    'message': 'Hello, world.'
  });
  grunt.registerTask('basic-task', 'This is an example task.', function() {
    grunt.log.writeln(grunt.config('basic-task.message'));
  });

  grunt.registerTask('default', 'This is the default task.', ['basic-task']);
};
```

要为任务设置描述, 只要在调用 `registerTask()` 时多传入一个参数即可。如果在命令行中请求帮助信息, Grunt 就会将这些任务描述打印出来。清单 2-11 截取了其中的一个片段。

清单 2-11 在命令行中请求帮助信息

```
$ grunt --help
...
Available tasks
  basic-task This is an example task.
  default This is the default task.
...
```

2.4.3 异步任务

默认情况下, Grunt 任务是同步执行的。只要任务函数返回, 即可认为任务已经执行完毕。然而, 有时候我们需要在任务中使用异步函数, 而且必须等待其执行完毕才能将控制权交还给 Grunt。清单 2-12 中展示了这个问题的解决方法。在任务中调用 `async()` 方法将通知 Grunt 此任务是异步执行的。该方法返回一个回调函数, 用于在任务完成时调用。在此之前, Grunt 暂不会执行任何额外的任务。

清单 2-12 异步 Grunt 任务

```
// example-async/tasks/list-files.js
```

```
var glob = require('glob');
```

```

module.exports = function(grunt) {
  grunt.registerTask('list-files', function() {
    /**
     * Grunt 将一直等待我们调用 done()函数来通知它异步任务执行完毕。
     */
    var done = this.async();
    glob('*', function(err, files) {
      if (err) {
        grunt.fail.fatal(err);
      }
      grunt.log.writeln(files);
      done();
    });
  });
};

```

2.4.4 任务依赖

对于复杂的 Grunt 工作流程,我们最好将其看作由一系列协同工作以达成最终结果的任务步骤组成。这种情况下,为任务指定一个或多个其他任务作为先决条件往往很有帮助,如清单 2-13 所示。

清单 2-13 声明任务依赖

```

// example-task-dependency/tasks/step-two.js

module.exports = function(grunt) {
  grunt.registerTask('step-two', function() {
    grunt.task.requires('step-one');
  });
};

```

本例中,step-two 任务要求 step-one 必须在其之前运行。任何试图直接调用 step-two 的行为都将导致错误发生,如清单 2-14 所示。

清单 2-14 依赖项运行之前直接运行任务导致 Grunt 报错

```

$ grunt step-two
Running "step-two" task
Warning: Required task "step-one" must be run first. Use --force to continue.

Aborted due to warnings.

```

2.4.5 多任务

除基本任务之外,Grunt 还支持“多任务”(multi-task)。多任务差不多是 Grunt 里最复杂的概念,所以如果你一开始感到困惑,别担心,这很正常。事实上,只要看过几个例子之后,它们的用途就会变得清晰起来,你也会从此踏上精通 Grunt 的道路。继续之前,还是让我们先看一个比较简单多任务示例及其配置(见清单 2-15)。

清单 2-15 Grunt 多任务

```

// example-list-animals/tasks/list-animals.js

module.exports = function(grunt) {
  /**

```



```

    * Our multi-task's configuration object. In this example, 'mammals'
    * and 'birds' each represent what Grunt refers to as a 'target.'
    */
    grunt.config('list-animals', {
      'mammals': {
        'animals': ['Cat', 'Zebra', 'Koala', 'Kangaroo']
      },
      'birds': {
        'animals': ['Penguin', 'Sparrow', 'Eagle', 'Parrot']
      }
    });

    grunt.registerMultiTask('list-animals', function() {
      grunt.log.writeln('Target:', this.target);
      grunt.log.writeln('Data:', this.data);
    });
};

```

多任务的使用极其灵活，其设计目的就是在单个项目中支持多种配置（称作“目标”，targets）。清单 2-15 中的多任务有两个目标：mammals 和 birds。如清单 2-16 所示，该任务可以按照任一目标运行。

清单 2-16 按照特定目标运行清单 2-15 中的任务

```

$ grunt list-animals:mammals
Running "list-animals:mammals" (#list-animals) task
Target: mammals
Data: { animals: [ 'Cat', 'Zebra', 'Koala', 'Kangaroo' ] }

Done, without errors.

```

运行多任务时也可以不传入任何参数。这种情况下，任务会为每个可用的目标都运行一次。清单 2-17 展示了在不指定目标的情况下运行多任务所产生的结果。

清单 2-17 在不指定目标的情况下，运行清单 2-15 中的多任务

```

$ grunt list-animals
Running "list-animals:mammals" (#list-animals) task
Target: mammals
Data: { animals: [ 'Cat', 'Zebra', 'Koala', 'Kangaroo' ] }

Running "list-animals:birds" (#list-animals) task
Target: birds
Data: { animals: [ 'Penguin', 'Sparrow', 'Eagle', 'Parrot' ] }

```

上述示例中，我们的多任务运行了两次，每个目标一次（mammals 和 birds）。注意在清单 2-15 中，我们的多任务引用了两个属性：this.target 和 this.data。这些属性使得多任务能够获取与当前正在运行的目标相关的信息。

2.4.6 多任务选项

在配置多任务时，任何存储在 options 键下的值（见清单 2-18）都会受到特殊处理。

清单 2-18 Grunt 多任务与 Options 配置项

```

// example-list-animals-options/tasks/list-animals.js

module.exports = function(grunt) {
  grunt.config('list-animals', {
    'options': {

```

```

    'format': 'array'
  },
  'mammals': {
    'options': {
      'format': 'json'
    },
    'animals': ['Cat', 'Zebra', 'Koala', 'Kangaroo']
  },
  'birds': {
    'animals': ['Penguin', 'Sparrow', 'Eagle', 'Parrot']
  }
});

grunt.registerMultiTask('list-animals', function() {
  var options = this.options();

  switch (options.format) {
    case 'array':
      grunt.log.writeln(this.data.animals);
      break;
    case 'json':
      grunt.log.writeln(JSON.stringify(this.data.animals));
      break;
    default:
      grunt.fail.fatal('Unknown format: ' + options.format);
      break;
  }
});
};

```

多任务选项为开发者提供了一种机制，即为任务定义和全局选项可以被目标任务的选项覆写。本例中，动物列表的全局选项 `format` 在任务层级被定义为 `'array'`。目标 `mammals` 将其覆写为 `'json'`，但是目标 `birds` 没有。这样，`mammals` 将显示为 JSON 串，而 `birds` 则继承全局选项的设置仍然显示为数组。

今后你遇到的绝大多数 Grunt 插件都会是可配置的多任务。由此而来的灵活性使得我们可以针对不同的应用环境，以不同的方式运行同一个任务。一个常见的情景就是为不同的构建环境创建不同的输出目标。例如，在编译程序时，针对本地开发环境和产品发布可以选用不同的任务运行配置。

2.4.7 模板配置

Grunt 配置对象支持配置嵌入式的模板字符串，可用于之后的其他配置。Grunt 支持的模板格式遵循 `Lodash` 和 `Underscore`，具体细节会涵盖在后续的章节中。清单 2-19 和清单 2-20 中的示例显示了该功能可以如何使用。

清单 2-19 Gruntfile 样例，Grunt 配置对象的 `pkg` 键中保存了项目 `package.json` 的内容

```

// example-templates/Gruntfile.js

module.exports = function(grunt) {
  grunt.initConfig({
    'pkg': grunt.file.readJSON('package.json')
  });
  grunt.loadTasks('tasks');
  grunt.registerTask('default', ['test']);
};

```

清单 2-20 使用自身配置的后续加载任务，可以使用模板来引用其他配置

```
// example-templates/tasks/test.js

module.exports = function(grunt) {
  grunt.config('test', {
    'banner': '<%= pkg.name %>-<%= pkg.version %>'
  });
  grunt.registerTask('test', function() {
    grunt.log.writeln(grunt.config('test.banner'));
  });
};
```

清单 2-19 展示了 Gruntfile 配置样例，其通过使用一系列与文件系统交互的内置方法，实现了加载项目 Package.Json 文件的内容，这些内置方法会在本章后续内容中讨论。文件内容随后存储在 Grunt 配置对象的 pkg 属性中。在清单 2-20 中可以看到，通过使用配置模板，任务能够直接引用配置对象 pkg 的信息。

2.4.8 命令行选项

通过如下格式可以为 Grunt 传递额外配置项。

```
$ grunt count --count=5
```

清单 2-21 中的例子展示了 Grunt 任务是如何通过 `grunt.option()` 方法获取信息的。从清单 2-22 中可以看到调用该任务的结果。

清单 2-21 简单 Grunt 任务计数到指定数字

```
// example-options/tasks/count.js

module.exports = function(grunt) {
  grunt.registerTask('count', function() {
    var limit = parseInt(grunt.option('limit'), 10);
    if (isNaN(limit)) grunt.fail.fatal('A limit must be provided (e.g. --limit=10)');
    console.log('Counting to: %s', limit);
    for (var i = 1; i <= limit; i++) console.log(i);
  });
};
```

清单 2-22 执行清单 2-21 中任务的输出结果

```
$ grunt count --limit=5
Running "count" task
Counting to: 5
1
2
3
4
5

Done, without errors.
```

2.4.9 提供反馈

为了在执行任务时为用户提供反馈信息，Grunt 提供了一些内置方法，其中某些方法你已经在本章中看过。当然，我们不会在此处列出所有方法，但表 2-1 中列出了一些常用的。

表 2-1

为用户显示反馈信息的实用 Grunt 方法

方法	描述
<code>grunt.log.write()</code>	输出消息到控制台
<code>grunt.log.writeln()</code>	输出消息到控制台，并在最后添加换行符
<code>grunt.log.oklns()</code>	输出表示成功的消息到控制台，并在最后添加换行符
<code>grunt.log.error()</code>	输出表示失败的消息到控制台，并在最后添加换行符
<code>grunt.log.subhead()</code>	输出表示强调的消息到控制台，并在最后添加换行符
<code>grunt.log.debug()</code>	输出消息到控制台，（仅当传入 <code>--debug</code> 标识时）

2.4.10 错误处理

在任务执行的过程中，会遇到错误。当遇到这种情况时，知道如何合理地处理它们是很重要的。当面对一个错误时，开发者需要使用 Grunt 的错误 API。它很易于使用，因为它只提供了两个方法（见表 2-2）。

表 2-2

Grunt 的错误 API 提供的可用方法

方法	描述
<code>grunt.fail.warn()</code>	显示警告并立即终止 Grunt
<code>grunt.fail.fatal()</code>	显示警告并立即终止 Grunt

2.5 操作文件系统

可想而知，作为构建工具，大部分 Grunt 插件都要以某种方式和文件系统交互。鉴于操作文件的重要性，Grunt 提供了有益的抽象允许开发者与文件系统交互，并且仅仅使用很少数量的样板代码。

当然，我们不会把所有方法都列举在此，表 2-3 显示了 Grunt 文件操作 API 中一些使用频率最高的方法。

表 2-3

与文件系统交互的实用 Grunt 方法

方法	描述
<code>grunt.file.read()</code>	读取和返回文件内容
<code>grunt.file.readJSON()</code>	读取文件内容，解析数据并作为 JSON，返回结果
<code>grunt.file.copy()</code>	把特定内容写入文件，如果需要，创建中间目录
<code>grunt.file.delete()</code>	删除特定文件路径
<code>grunt.file.mkdir()</code>	创建目录，创建任何可能没有的中间目录
<code>grunt.file.recurse()</code>	递归进入目录，为发现的每个文件执行回调

2.5.1 源-目标映射

许多 Grunt 任务与文件系统交互是依赖于源-目标映射的，该映射描述了要被处理的文件及各文件对应的目标。构建这样的映射会很冗长乏味，感谢 Grunt 为我们提供了解决此问题的有益捷径。

假设你正在开发一个项目，该项目根目录有一个公共文件夹。一旦工程部署后，这个文件夹中的文件要通过网络提供服务（见清单 2-23）。

清单 2-23 虚构项目的公共文件夹内容

```
// example-iterate1
```

```
└─ public
   └─ images
      ├── cat1.jpg
      ├── cat2.jpg
      └── cat3.png
```

如你所见，项目包含一个图片文件夹，其中有三个文件。了解这些内容之后，让我们看看 Grunt 能通过哪些方式来帮我们迭代这些文件。

如清单 2-24 所示，Grunt 的多任务配置和刚才介绍的很相似。关键不同之处在于任务配置中的 `src` 属性。在我们的任务中，Grunt 提供了一个 `this.files` 属性，该属性提供了一个包含着匹配到每个文件路径的数组，这些文件通过 `node-glob` 模块取得匹配到的文件路径数组。任务输出的结果可以在清单 2-25 中看到。

清单 2-24 Grunt 多任务配置对象，包含一个 `src` 键

```
// example-iterate1/tasks/list-files.js
```

```
module.exports = function(grunt) {
  grunt.config('list-files', {
    'images': {
      'src': ['public/**/*.jpg', 'public/* * /*.png']
    }
  });

  grunt.registerMultiTask('list-files', function() {
    this.files.forEach(function(files) {
      grunt.log.writeln('Source:', files.src);
    });
  });
};
```

清单 2-25 清单 2-24 中 Grunt 任务输出的结果

```
$ grunt list-files
Running "list-files:images" (#list-files) task
Source: [ 'public/images/cat1.jpg',
  'public/images/cat2.jpg',
  'public/images/cat3.png' ]
```

```
Done, without errors.
```

配置属性 `src` 和多任务属性 `this.files` 的结合使用，在迭代多文件时给开发者提供了精确的语法。以上的例子明显设计得非常简单，不过 Grunt 也同样为更复杂的场景提供额外选项。下面让我们来看一下。

与清单 2-24 中所示的键 `src` 的用法相反，清单 2-26 的例子展示了文件数组的使用——尽管这种方法有点冗长，但它为选择文件提供了更强大的数据格式。该格式接受额外的选项，从而让我们能够更好地调整我们的选择。特别重要的是它的 `expand` 选项，如清单 2-27 所示。请仔细关注，由于使用了 `expand` 选项，该示例与清单 2-26 的输出有所不同。

清单 2-26 使用“文件数组”格式迭代文件

```
// example-iterate2/tasks/list-files.js
```

```
module.exports = function(grunt) {
```

```

grunt.config('list-files', {
  'images': {
    'files': [
      {
        'cwd': 'public',
        'src': ['**/*.jpg', '**/*.png'],
        'dest': 'tmp',
        'expand': true
      }
    ]
  }
});

grunt.registerMultiTask('list-files', function() {
  this.files.forEach(function(files) {
    grunt.log.writeln('Source:', files.src);
    grunt.log.writeln('Destination:', files.dest);
  });
});

```

清单 2-27 清单 2-26 中 Grunt 任务的输出

```

$ grunt list-files
Running "list-files:images" (#list-files) task
Source: [ 'public/images/cat1.jpg' ]
Destination: tmp/images/cat1.jpg
Source: [ 'public/images/cat2.jpg' ]
Destination: tmp/images/cat2.jpg

Done, without errors.

```

当该 `expand` 选项与 `dest` 选项成对出现的时候，表示明 Grunt 会迭代访问任务中的 `this.files.forEach` 循环所找到的每个入口文件。在每个文件内，我们可以找到对应的 `dest` 属性。使用这种方法，我们就能很容易地创建源-目的地映射，可以用于把文件从一个位置复制（或移动）到另一个位置。

2.5.2 监视文件变化

当特定模式下匹配到的文件被创建、修改、删除时，Grunt 最为流行的插件之一 `grunt-contrib-watch` 使 Grunt 有能力运行预定义的任务。当和其他任务结合在一起时，`grunt-contrib-watch` 让开发者能创建强大的工作流，用于自动执行如下工作：

- 检查 JavaScript 代码中的错误（如静态语法分析“linting”）；
- 编译 Sass/L 样式表；
- 运行单元测试

下面让我们看几个实践中运用上述工作流的例子。

1. JavaScript 自动化静态检查

清单 2-28 显示了一个基本的 Grunt 设置，和本章已经展示的示例相似。监视任务以默认任务作为别名注册在 Grunt 中，这样我们就能简单地在命令行中运行 `$ grunt` 来监视项目中的变化。本例中，Grunt 会监视 `src` 文件夹下的变化。一旦变化发生，`jshint` 任务就会触发。

清单 2-28 发生变化时自动检查 JavaScript 错误

```
// example-watch-hint/Gruntfile.js
```

```

module.exports = function(grunt) {
    grunt.loadTasks('tasks');
    grunt.registerTask('default', ['watch']);
};

// example-watch-hint/tasks/jshint.js

module.exports = function(grunt) {

    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.config('jshint', {
        'options': {
            'globalstrict': true,
            'node': true,
            'scripturl': true,
            'browser': true,
            'jquery': true
        },
        'all': [
            'src/**/*.js'
        ]
    });
};

// example-watch-hint/tasks/watch.js

module.exports = function(grunt) {

    grunt.loadNpmTasks('grunt-contrib-watch');

    grunt.config('watch', {
        'js': {
            'files': [
                'src/**/*.js'
            ],
            'tasks': ['jshint'],
            'options': {
                'spawn': true
            }
        }
    });
};

```

2. 自动编译 Sass 样式表

清单 2-29 展示了这样一个示例，它引导 Grunt 来监视项目变化。然而这次不同于监视 JavaScript 文件，Grunt 监视的是项目的 Sass 样式表。当变化发生的时候，插件 `grunt-contrib-compass` 被调用，把样式表编译为最终形态。

清单 2-29 变化发生时自动编译 Sass 样式表

```

// example-watch-sass/Gruntfile.js

module.exports = function(grunt) {
    grunt.loadTasks('tasks');
    grunt.registerTask('default', ['watch']);
};

// example-watch-sass/tasks/compass.js

```

```

module.exports = function(grunt) {
    grunt.loadNpmTasks('grunt-contrib-compass');
    grunt.config('compass', {
        'all': {
            'options': {
                'httpPath': '/',
                'cssDir': 'public/css',
                'sassDir': 'scss',
                'imagesDir': 'public/images',
                'relativeAssets': true,
                'outputStyle': 'compressed'
            }
        }
    });
};

// example-watch-compass/tasks/watch.js

module.exports = function(grunt) {
    grunt.loadNpmTasks('grunt-contrib-watch');
    grunt.config('watch', {
        'scss': {
            'files': [
                'scss/**/*.scss'
            ],
            'tasks': ['compass'],
            'options': {
                'spawn': true
            }
        }
    });
};

```

注意 若想让本例生效，前提是需要安装 Compass。它是一个开源 CSS 编辑框架。可以在网站 <http://compass-style.org/install> 上了解更多信息。

3. 自动化单元测试

最后一个例子是使用 `grunt-contrib-watch` 插件来完成单元测试。清单 2-30 展示了用于监视 JavaScript 变化的 Gruntfile。变化发生时将会立即触发项目的单元测试，单元测试由 `grunt-mocha-test` 插件完成。

清单 2-30 当变化发生时自动运行单元测试

```

// example-watch-test/Gruntfile.js

module.exports = function(grunt) {
    grunt.loadTasks('tasks');
    grunt.registerTask('default', ['watch']);
};

// example-watch-test/tasks/mochaTest.js

```



```

module.exports = function(grunt) {

  grunt.loadNpmTasks('grunt-mocha-test');

  grunt.config('mochaTest', {
    'test': {
      'options': {
        'reporter': 'spec'
      },
      'src': ['test/**/*.js']
    }
  });
};

```

```
// example-watch-test/tasks/watch.js
```

```

module.exports = function(grunt) {

  grunt.loadNpmTasks('grunt-contrib-watch');

  grunt.config('watch', {
    'scss': {
      'files': [
        'src/**/*.js'
      ],
      'tasks': ['mochaTest'],
      'options': {
        'spawn': true
      }
    }
  });
};

```

2.6 创建 Grunt 插件

社区提供的丰富插件库是让 Grunt 真正变得闪耀的库，它能使你立即从 Grunt 中获益，而不是需要从头创建复杂的任务。如果你需要在项目中做自动构建，那么很可能某人已经为你做好这项“Grunt”工作。在这一节中，你可以懂得如何向社区回馈自己创建的 Grunt 插件。

2.6.1 开始

首先要做的事情之一是创建一个公共的 GitHub 仓库，以存储你的新插件。下文中将要提及的示例包含在本书附带的源码中，本书附带了源码。一旦你准备好代码仓库，就把它克隆到你的电脑上。下一步，按照本章前面“将 Grunt 添加到项目中”一节所概述的步骤，在仓库目录中初始化 Grunt。然后，你的新 Grunt 插件的文件模式将会类似清单 2-31 中的示例。

清单 2-31 新 Grunt 插件的文件模式

```

├── Gruntfile.js
├── README.md
├── package.json
└── tasks

```

■ **注意** 这里提到的最重要一点是创建 Grunt 插件不需要任何额外的模式或知识（抛开本章已经涵盖的内容不说）。这个过程反映了把 Grunt 整合进一个现有的项目中，Gruntfile 的建立用于加载任务，除此之外是任务本身。一旦插件发布至 npm，其他的 Grunt 项目就能够像本章中到处提及的方式一样来加载你的插件。

2.6.2 创建任务

按照示例中的方式，让我们创建一个 Grunt 插件，该插件能够生成一份描述了项目中包含的文件类型、大小和数量的报告。该插件的配置示例如清单 2-32 所示。

清单 2-32 插件配置示例代码

```
// example-plugin/Gruntfile.js

module.exports = function(grunt) {

  grunt.config('file-report', {
    'options': {
    },
    'public': {
      'src': ['public/**/*']
    },
    'images': {
      'src': ['public/**/*.jpg', 'public/*/*.png', 'public/**/*.gif']
    }
  });

  grunt.loadNpmTasks('grunt-file-reporter');
  grunt.registerTask('default', ['file-report']);
};
```

从清单 2-33 中可以看到插件的源码，其中 Grunt 注册了一个多任务 **file-report**。每当调用这个任务，清单 2-32 中指定的目标文件会被迭代遍历。完成这个任务后，插件会编译得到一个报告展示出它发现的文件类型、数量和大小详情。

清单 2-33 插件的源码

```
// example-plugin/node_modules/grunt-file-reporter/Gruntfile.js

var fs = require('fs');
var filesize = require('filesize');
var _ = require('lodash');
_.mixin(require('underscore.string'));

module.exports = function(grunt) {

  var mime = require('mime');
  var Table = require('cli-table');

  grunt.registerMultiTask('file-report', 'Generates a report of file types & sizes used within a project ', function() {

    var report = {
      'mimeTypes': {},
      'largest': null,
      'smallest': null
    };
  });
};
```

```

    });

    var table = new Table({
      'head': ['Content Type', 'Files Found', 'Total Size',
        'Average Size', 'Largest', 'Smallest']
    });
    var addFile = function(file) {
      if (grunt.file.isDir(file)) return;
      var mimeType = mime.lookup(file);
      if (!report.mimeTypes[mimeType]) {
        report.mimeTypes[mimeType] = {
          'count': 0,
          'sizes': [],
          'largest': null,
          'smallest': null,
          'oldest': null,
          'newest': null
        };
      }
      var details = report.mimeTypes[mimeType];
      details.count++;
      var stats = fs.statSync(file);
      details.sizes.push(stats.size);
      if (!details.largest || stats.size > details.largest.size) {
        details.largest = { 'file': file, 'size': stats.size };
      }
      if (!report.largest || stats.size > report.largest.size) {
        report.largest = { 'file': file, 'size': stats.size };
      }
      if (!details.smallest || stats.size < details.smallest.size) {
        details.smallest = { 'file': file, 'size': stats.size };
      }
      if (!report.smallest || stats.size < report.smallest.size) {
        report.smallest = { 'file': file, 'size': stats.size };
      }
    };
    var sum = function(arr) {
      return arr.reduce(function(a, b) {
        return a + b;
      });
    };

    var displayReport = function() {
      var totalSum = 0;
      var totalFiles = 0;
      var totalSizes = [];
      _each(report.mimeTypes, function(data, mType) {
        var fileSum = sum(data.sizes);
        totalSum += fileSum;
        totalFiles += data.sizes.length;
        totalSizes = totalSizes.concat(data.sizes);
        table.push([mType, data.count, filesize(fileSum),
          filesize(fileSum / data.sizes.length),
          _sprintf('%s (%s)', data.largest.file, filesize(data.largest.size)),
          _sprintf('%s (%s)', data.smallest.file, filesize(data.smallest.size))],
        );
      });
      table.push(['-', totalFiles, filesize(totalSum),
        filesize(totalSum / totalSizes.length),
        _sprintf('%s (%s)', report.largest.file, filesize(report.largest.size)),
        _sprintf('%s (%s)', report.smallest.file, filesize(report.smallest.size))],
    );
  };

```

```

        console.log(table.toString());
    };
    this.files.forEach(function(files) {
        files.src.forEach(addFile);
    });
    displayReport();
});
};

```

由 file-report 插件生成的输出如图 2-1 所示。

Running "file-report:public" (file-report) task

Content Type	Files Found	Total Size	Average Size	Largest	Smallest
image/jpeg	6	957.84 kB	159.64 kB	public/images/desperate-shatner.jpg (377.68 kB)	public/images/khaaon.jpg (13.82 kB)
image/png	2	1.88 MB	964.98 kB	public/images/cat-tongue.png (1.65 MB)	public/images/cat-hat.png (239.56 kB)
image/gif	1	947.93 kB	947.93 kB	public/images/fresh-prince.gif (947.93 kB)	public/images/fresh-prince.gif (947.93 kB)
text/html	1	289 B	289 B	public/index.html (289 B)	public/index.html (289 B)
-	10	3.75 MB	383.6 kB	public/images/cat-tongue.png (1.65 MB)	public/index.html (289 B)

Running "file-report:images" (file-report) task

Content Type	Files Found	Total Size	Average Size	Largest	Smallest
image/jpeg	6	957.84 kB	159.64 kB	public/images/desperate-shatner.jpg (377.68 kB)	public/images/khaaon.jpg (13.82 kB)
image/png	2	1.88 MB	964.98 kB	public/images/cat-tongue.png (1.65 MB)	public/images/cat-hat.png (239.56 kB)
image/gif	1	947.93 kB	947.93 kB	public/images/fresh-prince.gif (947.93 kB)	public/images/fresh-prince.gif (947.93 kB)
-	9	3.75 MB	426.19 kB	public/images/cat-tongue.png (1.65 MB)	public/images/khaaon.jpg (13.82 kB)

Done, without errors.
mbp:example-plugin tim\$

图 2-1 file-report 任务所生成的输出

2.6.3 将任务发布到 npm

一旦我们的插件已经就绪并且我们的 Git 仓库也更新到最新的代码，最后一步就是通过 npm 发布插件使他人也可用。

```
$ npm publish
```

注意 如果这是你第一次向 npm 发布模块，你将被要求创建一个账号。

2.7 小结

本章中，我们学习了功能强大的 Grunt 套件，让开发者对重复、枯燥冗长的任务进行自动化处理，还展示了：

- Grunt 的工作原理（任务、插件和配置对象）。
- 如何配置任务和插件。
- 如何利用 Grunt 内置的很多实用工具为用户提供反馈信息、与文件系统交互。
- 如何创建和分享你自己的插件。

2.8 相关资源

- Grunt: <http://gruntjs.com>
- JSHint: <http://jshint.com>
- grunt-contrib-watch: <https://github.com/gruntjs/grunt-contrib-watch>
- grunt-contrib-jshint: <https://github.com/gruntjs/grunt-contrib-jshint>
- grunt-contrib-uglify: <https://github.com/gruntjs/grunt-contrib-uglify>
- grunt-contrib-compass: <https://github.com/gruntjs/grunt-contrib-compass>
- grunt-mocha-test: <https://github.com/pghalliday/grunt-mocha-test>
- Syntactically Awesome Stylesheets (Sass): <http://sass-lang.com>
- Compass: <http://compass-style.org>

清单 3-2 使用 modernweb 脚手架创建我们的第一个项目

```
$ mkdir my-app
$ cd my-app
$ yo modernweb
```

```
? Project file: my-app
? Project name: my-app
? Project description: my-app
? Project author: John Doe
? Project license: MIT
```

在回答完脚手架的几个问题（你可以安全地接受默认值）之后，Yeoman 将开始创建项目。然后，我们可以使用项目默认值（脚手架已经帮我们设置好了），这些都已经包含在 Yeoman 的默认配置中（见清单 3-3）。

清单 3-3 使用 Yeoman 脚手架创建我们的第一个项目

```
$ grunt
Running "compass:dist" task
File public/css/main.css created.

Running "uglify:dist" task
Unchanged.

Running "mocha:dist" task
Unchanged.

Running "mocha:dist" task
Unchanged.
```

清单 3-1 通过 npm 安装 Yeoman

```
$ npm install -g yo
$ yo --version
0.1.1
```

Yeoman

生活里人仅需两样东西——润滑剂和胶布，事物运行始于前者而止于后者。

——威尔阿彻·格林

近几年，开发者社区目睹了一次角色上排序的转变。Web 应用程序相较于原生的应用一度被认为是二等公民，现在却在很大程度上取代了传统的桌面应用，这得益于现代化 Web 开发技术的广泛采用以及移动网络的兴起。但随着 Web 应用的日渐复杂，Web 应用所依赖的工具和其引导程序也开始日渐发展起来。

本章所要探讨的 Yeoman，是当下一个流行的项目构建的脚手架工具，通过自动化处理一些乏味的任务来帮助新应用从 0 开始顺利构建。Yeoman 为了创建可复用的模板，提供了一种机制来描述项目的初始文件结构、HTML 文件、第三方库依赖以及任务处理器的配置。这些模板可以通过 npm 与更活跃的开发者社区共享，使开发人员能够快速构建遵循广泛认可的最佳实践的应用。

在本章中，你将学会如何：

- 安装 Yeoman
- 使用已经在社区发布的 Yeoman 脚手架
- 用自己的 Yeoman 脚手架来回馈 Yeoman 社区

■ 注意 本章建立在前面讲述的 Bower 和 Grunt 的章节的基础上。如果你不熟悉这些工具，希望你继续阅读本章之前先了解这两章的内容。

3.1 安装 Yeoman

Yeoman 的命令行工具 yo 可以通过 npm 安装。如果你还没有安装 Yeoman，可以按照如下方式安装（见清单 3-1）。

清单 3-1 通过 npm 安装 yo 命令行工具

```
$ npm install -g yo
$ yo --version
1.4.6
```

3.2 创建第一个项目

Yeoman 允许开发者使用被 Yeoman 称作“脚手架”的这类工具，通过可复用的模板来快速创建

应用程序的初始结构。为了更好地理解此过程是如何改进你的工作流程的，我们使用专门为本章创建的一个名为 `modernweb` 的脚手架来创建一个新的项目。然后，我们看看这个脚手架如何根据你的需求被创建，以及如何在开发者社区中分享你的自定义 `Yeoman` 脚手架。

该项目用到以下工具和库。

- Grunt
- Bower
- jQuery
- AngularJS
- Browserify
- Compass

`Yeoman` 脚手架被安装成全局模块。在这种情况下，我们的脚手架的安装命令看起来应该这样：

```
$ npm install -g generator-modernweb
```

■ 注意 这个脚手架的前缀是 `generator-`，这是所有的 `Yeoman` 脚手架必须遵守的重要约定。在运行时，`Yeoman` 将通过搜索全局安装的并且匹配这个命名规范的模块来确定哪些（如果有）脚手架已经安装。

随着我们脚手架的安装，我们可以进行下一步，建立我们的第一个项目。第一步，我们创建一个新的文件夹来包含这个项目。然后，让 `Yeoman` 通过我们刚刚安装的脚手架来创建一个新的项目。清单 3-2 展示了伴随脚手架设计好的几个提示问题来创建的步骤。

清单 3-2 使用 `modernweb` 脚手架创建我们的第一个项目

```
$ mkdir my-app
$ cd my-app
$ yo modernweb

? Project Title: My Project
? Package Name: my-project
? Project Description: My awesome project
? Project Author: John Doe
? Express Port: 7000
```

在回答完脚手架的几个问题（你可以安全地接受默认值）之后，`Yeoman` 将开始创建项目。然后，我们可以使用项目默认的 `Grunt` 工具轻松地编译和启动项目，这些都被脚手架很轻松地设置好了（见清单 3-3）。

清单 3-3 我们新项目默认的 `Grunt` 任务将触发许多编译步骤，并且在浏览器中运行该项目

```
$ grunt
Running "concat:app" (concat) task
File public/dist/libs.js created.

Running "compass:app" (compass) task
unchanged scss/style.scss
Compilation took 0.002s

Running "browserify" task

Running "concurrent:app" (concurrent) task
```

```
Running "watch" task
Waiting...
Running "open:app" (open) task
Running "server" task
Server is now listening on port: 7000
```

Done, without errors.

如你所见，项目默认的 Grunt 任务执行了几个附加的编译步骤：

- 多个 JavaScript 库被编译成一个经过压缩的脚本文件。
- Sass 样式表文件被编译（成 css 文件）。
- 应用的源码被 Browserify 工具编译。
- 创建一个 Express 实例来提供服务。
- 初始化监听脚本，这种监听会在我们修改文件之后自动地重新编译我们的项目。

我们程序默认的 Grunt 任务最后一步是启动程序并且在浏览器中打开，如图 3-1 所示。

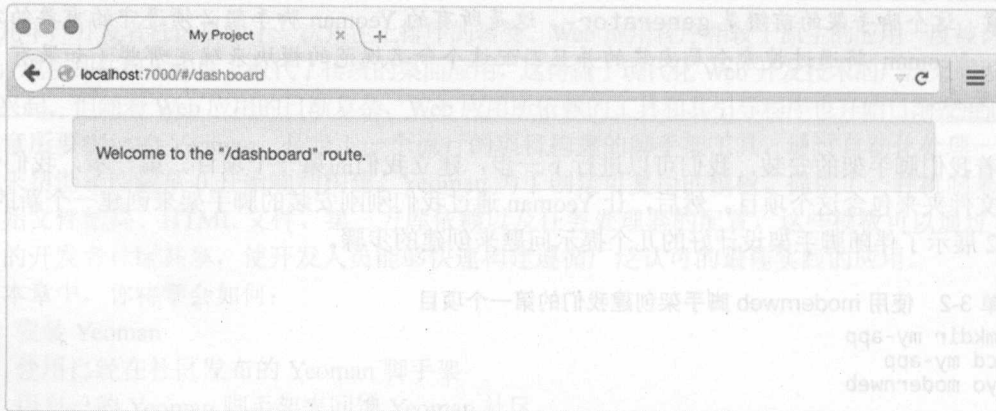


图 3-1 默认 Grunt 任务打开的项目首页

现在，新项目准备好要进一步开发，让我们花一些时间来熟悉各种模板、脚本和 Grunt 任务。我们的脚手架已经准备就绪。对我们来说，要特别注意这些文件的内容：

- bower.json
- Gruntfile.js
- package.json
- public/index.html

在 Yeoman 工具的用户提示和模板（我们将在下一节详细讨论）的帮助下，脚手架已经将一些我们不知道如何填写的内容写在文件中，如 package.json 文件中的 name、description 和 author 的值。

清单 3-4 package.json 文件的内容

// package.json

```
{
  "name": "my-project",
  "description": "My awesome project",
  "author": "John Doe",
  "files": [],
```



```

"keywords": [],
"dependencies": {},
"browserify": {
  "transform": [
    "brfs",
    "bulkify",
    "folderify"
  ],
},
"browser": {}
}

```

子命令

脚手架作为可配置的模板来简化新项目的创建过程只是最简单的应用方式，并不是唯一的目的。除了协助新项目的搭建，脚手架也会包含一些项目创建者在开发过程中沉淀的其他有用的命令。

在清单 3-2 中，我们使用 **modernweb** 脚手架创建了一个使用 **AngularJS** 框架的单页应用。如果你对 **Angular** 并不熟悉，先别急——目前这个框架的细节对我们来说并不重要。然而，重要的是项目的 **public/app/routes** 这个文件夹的内容。注意，这个路径下已经为我们创建了一个单独的文件夹 **dashboard**。这个文件夹的内容如清单 3-5 所示。

清单 3-5 public/app/routes/dashboard 文件夹的内容

```

├── index.js
└── template.html

```

// public/app/routes/dashboard/index.js

```

module.exports = {
  'route': '/dashboard',
  'controller': function() {
  },
  'templateUrl': '/app/routes/dashboard/template.html',
  'resolve': {}
};

```

// public/app/routes/dashboard/template.html

```

<div class="well">
  Welcome to the "/dashboard" route.
</div>

```

项目已经在 **public/app/routes** 目录中创建了一些类似的文件夹，这些文件夹在应用中被定义成“hashbang”模式的路由。在本例中，项目的 **dashboard** 文件夹也是一个路由，可以用 **http://localhost:7000/#/dashboard** 地址来访问。知道了这一点，假设我们要为我们的应用添加一个 **users** 路由。我们可以手动地在合适的位置创建一些必要的文件。或者，我们也可以使用脚手架提供的附加命令来简化这个过程（见清单 3-6）。

清单 3-6 调用脚手架命令自动创建路由示例

```

$ yo modernweb:route users
  create public/app/routes/users/index.js

```

```
create public/app/routes/users/template.html
Route `users` created.
```

运行命令之后，项目的 `/public/app/routes` 文件夹里有一个新的文件夹是 `users`。在这个文件夹里，Yeoman 脚手架已经生成合适的文件。如果你仍然保留着我们在清单 3-3 中启动的服务，你会发现监听脚本已经监测到这个变化，并且为我们的应用自动地重新编译一次（见清单 3-7）。

清单 3-7 内容改变之后 Grunt 自动重新编译应用

```
>> File "public/app/routes/users" added.
Running "browserify" task
Done, without errors.
```

3.3 创建你的第一个脚手架

本章的后续部分将集中在如何创建自定义脚手架——与前面提到的类似的围绕 AngularJS（以及其他工具）来引导创建一个新项目相同。之后，你会做好准备开始创建自己的脚手架，这将使你快速安装并运行满足你特定需求的工作流。

3.3.1 Yeoman 脚手架是一个 Node 模块

Yeoman 脚手架不过是一个遵循 Yeoman 的规定准则的简单的 Node 模块。本质上，创建一个脚手架工具的第一步是创建一个新的 Node 模块。清单 3-8 展示了需要的命令以及生成的 `package.json` 文件。

清单 3-8 创建一个包含我们第一个 Yeoman 脚手架工具所需内容的 Node 模块

```
$ mkdir generator-example
$ cd generator-example
$ npm init

// generator-example/package.json
{
  "name": "generator-example",
  "version": "1.0.0",
  "description": "An example Yeoman generator",
  "files": [],
  "keywords": [
    "yeoman-generator"
  ],
  "dependencies": {}
}
```

注意 虽然我们遵循了与本章中用到的 `mordernweb` 脚手架相同的创建步骤，但我们仍要为新创建的模块指定一个不同的名称，从而避免与已经安装的模块冲突。还要注意，模块的关键词要包含 `yeoman-generator`。Yeoman 的网站维护了一个囊括 npm 中所有可用的脚手架列表，便于开发人员找到已有的脚手架来满足他们的需求。如果要从这个列表中引用一个脚手架，那么这个脚手架的 `package.json` 文件中必须包含这个 `yeoman-generator` 关键字，跟在 `package.json` 中它的描述项后面。

Yeoman 脚手架可以依赖外部模块，其方式与其他的 Node 模块一样。最起码，每个脚手架工具必须把 **yeoman-generator** 指定为一个本地依赖。这个模块将提供 Yeoman 提供的一些核心功能，创建与用户的交互，与文件系统交互或者其他重要的任务。这个模块会使用以下命令安装在本地依赖中。

```
$ npm install yeoman-generator --save
```

3.3.2 子脚手架

Yeoman 脚手架包含了不止一个命令，每个命令都可以通过命令行工具被单独调用。这些命令被 Yeoman 称为“子脚手架”，是在模块的根目录下存在的文件夹中定义的。对于一些额外的上下文，我们回头看清单 3-2，其中我们基于 **morderweb** 脚手架的 **\$ yo morderweb** 命令创建了一个新项目。在这个例子中，我们并没有指定一个命令，只是用到了脚手架的名字。因此，Yeoman 执行了一个脚手架的默认子命令，按照惯例被指定为 **app**（目录）。我们可以通过运行这个命令来完成同样的事情。

```
$ yo modernweb:app
```

为了更好地理解这是如何工作的，我们创建脚手架默认的 **app** 子命令，分为以下四步。

1. 在模块的根目录下创建一个名为 **app** 的文件夹。
2. 在 **app** 文件夹中创建 **templates** 文件夹。
3. 将我们想要复制到目标项目里的各种文件放入我们的 **templates** 文件夹中（比如 HTML 文件、Grunt 任务、Bower 依赖清单等）。
4. 创建如清单 3-9 所示的脚本，负责驱动该功能。

清单 3-9 脚手架默认 **app** 命令（子脚手架）的内容

```
// generator-example/app/index.js
```

```
var generators = require('yeoman-generator');
/**
 * 通过导出一个类来创建脚手架，该类继承自
 * Yeoman 的 'Base' 类。
 */

module.exports = generators.Base.extend({
  'prompting': function () {
    /**
     * 表示此函数会异步执行。Yeoman 会在继续运行前一直等待，直到调用
     * 了 'done()' 函数。
     */
    var done = this.async();
    /**
     * 脚手架的 'prompt' 方法（继承自 Yeoman 的 'Base'
     * 类）允许我们定义一系列问题来提示用户使用。
     */
    this.prompt([
      {
        'type': 'input',
        'name': 'title',
        'message': 'Project Title',
        'default': 'My Project',
        'validate': function (title) {
          return (title.length > 0);
        }
      },
    ],
```

```

    'type': 'input',
    'name': 'package_name',
    'message': 'Package Name',
    'default': 'my-project',
    'validate': function (name) {
      return (name.length > 0 && /^[a-z0-9-]+\$/i.test(name));
    },
    'filter': function (name) {
      return name.toLowerCase();
    }
  },
  {
    'type': 'input',
    'name': 'description',
    'message': 'Project Description',
    'default': 'My awesome project',
    'validate': function (description) {
      return (description.length > 0);
    }
  },
  {
    'type': 'input',
    'name': 'author',
    'message': 'Project Author',
    'default': 'John Doe',
    'validate': function (author) {
      return (author.length > 0);
    }
  },
  {
    'type': 'input',
    'name': 'port',
    'message': 'Express Port',
    'default': 7000,
    'validate': function (port) {
      port = parseInt(port, 10);
      return (!isNaN(port) && port > 0);
    }
  }
], function (answers) {
  this.answers = answers;
  done();
}.bind(this));
},
'writing': function () {
  /**
   * 从我们脚手架的'templates'文件夹复制文件到目标项目
   * 每个文件在写入磁盘之前，其内容都被视为 Lodash 模板来处理
   */
  this.fs.copyTpl(
    this.templatePath('**/*'),
    this.destinationPath(),
    this.answers
  );

  this.fs.copyTpl(
    this.templatePath('pkg.json'),
    this.destinationPath('package.json'),
    this.answers
  );

  this.fs.delete(this.destinationPath('pkg.json'));

  this.fs.copyTpl(

```



```

    this.templatePath('.bowerrc'),
    this.destinationPath('.bowerrc'),
    this._answers
  );
  /**
   * 向目标项目的文件夹中写 Yeoman 配置文件。
   */
  this.config.save();
},
'install': function () {
  /**
   * 安装项目文件夹中的各个 npm 模块并更新相应 'package.json'。
   */
  this.npmInstall([
    'express', 'lodash', 'underscore.string', 'browserify',
    'grunt', 'grunt-contrib-concat', 'grunt-contrib-watch',
    'grunt-contrib-compass', 'grunt-concurrent', 'bulk-require',
    'brfs', 'bulkify', 'folderify', 'grunt-open'
  ], {
    'saveDev': false
  });
  /**
   * 安装定义在 'bower.json' 中的依赖。
   */
  this.bowerInstall();
},
'end': function () {
  this.log('Your project is ready.');
```

我们脚手架的 app 文件夹内容如图 3-2 所示。

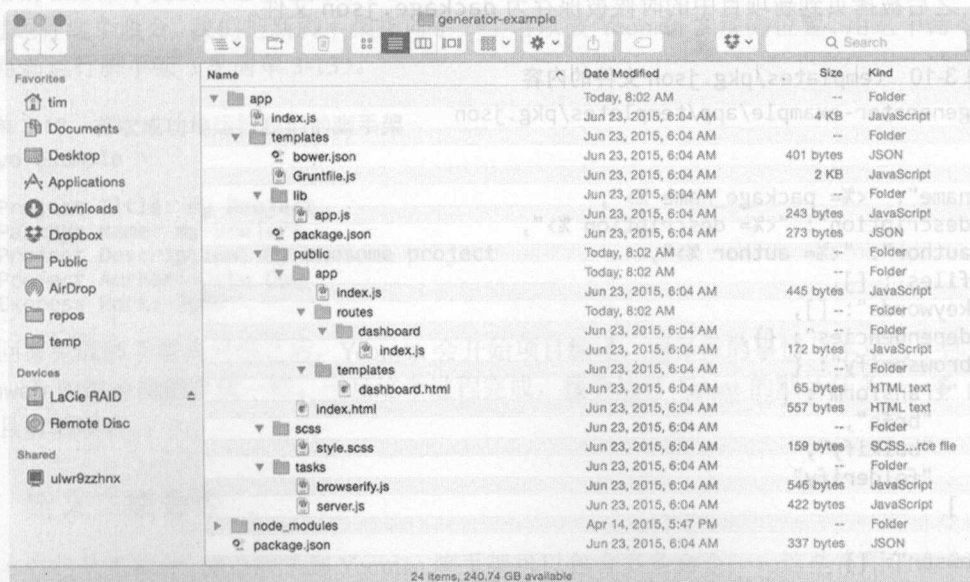


图 3-2 脚手架 app 文件夹的内容，templates 文件夹下的内容会被拷贝到目标工程里

在清单 3-9 中，我们脚手架默认的 **app** 命令是由继承自 **Yeoman** 基类的类创建的。在这个类中，定义了四个实例方法：

- **prompting()**
- **writing()**
- **install()**
- **end()**

这些方法在执行过程中起着重要作用（他们不是被任意选择）。脚手架运行的时候，它将搜索与下述列表相匹配的原型方法。

- **initializing()**：初始化方法（检查项目状态，获取配置）。
- **prompting()**：提示用户输入信息。
- **configuring()**：保存配置文件。
- **default()**：不包含在这个列表中的原型方法名称将在此步骤执行。
- **writing()**：脚手架的写操作发生在这里。
- **conflicts()**：冲突将在这里处理（在 **Yeoman** 内部使用）。
- **install()**：安装过程发生在这里（**npm**，**bower**）。
- **end()**：最后一个被调用的方法，用来清理、关闭消息。

一旦 **Yeoman** 编译了存在于我们脚手架中的各种原型方法的列表，它将按照上述列表中的优先级执行。

Lodash 模板

在清单 3-9 中，**Yeoman** 的 **fs.copyTpl()** 方法用来把子脚手架的模板文件夹拷贝到目标项目中。这个方法不同于 **Yeoman** 的 **fs.copy()** 方法，因为它会把每个文件视为 **Lodash** 模板文件。清单 3-10 展示了我们子脚手架的 **templates/pkg.json** 文件的内容。这个文件也被当作了 **Lodash** 的模板文件来处理，之后被拷贝到新项目中的时候被保存为 **package.json** 文件。

清单 3-10 templates/pkg.json 文件的内容

```
// generator-example/app/templates/pkg.json

{
  "name": "<%= package_name %>",
  "description": "<%= description %>",
  "author": "<%= author %>",
  "files": [],
  "keywords": [],
  "dependencies": {},
  "browserify": {
    "transform": [
      "brfs",
      "bulkify",
      "folderify"
    ]
  },
  "browser": {}
}
```

■ **注意** 在这个过程中，Yeoman 可以改变它们的行为，也可以根据用户的答案来改变模板内容，这给用户带来了很多人兴奋的可能性。用户可以根据他们的特别需求来定制新项目。Yeoman 的这一点，比起其他工具来说，显得更加有用。

我们现在准备用新的脚手架来创建第一个项目。首先，打开一个终端，创建一个文件夹来包含这个项目。下一步进入新的文件夹中，运行脚手架，如清单 3-11 所示。

清单 3-11 首次运行我们的新脚手架

```
$ mkdir new-project
$ cd new-project
$ yo example
```

Error example

You don't seem to have a generator with the name example installed.

You can see available generators with npm search yeoman-generator and then install the with npm install [name].

显然，这并不是我们想要的结果。为了弄明白是什么造成的错误，回想一下本章之前提到的 Yeoman 的调用方式，它会在本地全局安装的模块中查找以 **generator-**命名的模块。因此，Yeoman 目前尚且不知道我们创建的脚手架的存在。幸运的是，npm 为我们提供了一个方便的命令来解决这个问题。npm 的 npm link 命令在我们的新模块和 Node 的全局模块文件夹之间创建了一个符号连接（软链）。命令要在我们新模块的根目录执行（见清单 3-12）。

清单 3-12 使用 npm link 命令创建一个符号链接

```
$ npm link
/Users/tim/.npm/v0.10.33/lib/node_modules/generator-example -> /opt/generator-example
```

npm 的 link 命令在我们运行它的那个文件夹与全局的 Node 模块的文件夹之间建立了一个符号链接。通过运行这个命令，我们把新脚手架指向了一个可以被 Yeoman 发现的位置。用这个命令替换后，我们再重新运行脚手架（见清单 3-13）。

清单 3-13 首次成功地运行我们的脚手架

```
$ yo example

? Project Title: My Project
? Package Name: my-project
? Project Description: My awesome project
? Project Author: John Doe
? Express Port: 7000
```

在回答完成脚手架的问题之后，Yeoman 会开始项目编译，编译做的事情与本章前半部分用到的 modernweb 的编译做的事情一样。一旦这个过程完成，接着运行 Grunt 的默认任务——\$ grunt 来编译并且启动项目。

3.3.3 定义二级命令

在本章的前半部分，你已经学到 Yeoman 脚手架可以包含多条命令——这些常用的命令在初始项目的创建之外，也可以很好地延伸和扩展。modernweb 脚手架通过引入 route 命令展示了这一过程。

这个 `route` 命令主要执行在 Angular 应用中自动创建新路由过程（参考本章前面的清单 3-6）。创建此命令涉及的步骤紧随我们创建默认 `app` 命令的时候遵循的步骤。

1. 在脚手架模块根目录下创建一个名为 `route` 的文件夹。
2. 在新建的 `route` 文件夹下创建 `templates` 文件夹。
3. 将各种我们想复制到目标项目中的文件放入 `templates` 文件夹中。
4. 创建如清单 3-14 所示的脚本，负责驱动 `route` 命令。

清单 3-14 用 `route` 子脚手架创建新 Angular 路由

```
// generator-example/route/index.js
```

```
var generators = require('yeoman-generator');
```

```
/*
```

我们脚手架的默认 `'app'` 命令通过扩展自 Yeoman 的 `'Base'` 类来创建。本例中取而代之，我们扩展的是 `'NamedBase'` 类。这样做可提示 Yeoman 该命令期待接受一个或多个参数，如 `$ yo example:route`。

```
my-new-route
```

```
*/
```

```
module.exports = generators.NamedBase.extend({
```

```
  'constructor': function (args) {
    this._opts = {
      'route': args[0]
    };
    generators.NamedBase.apply(this, arguments);
  },
```

```
  'writing': function () {
```

```
    this.fs.copyTpl(
      this.templatePath('index.js'),
      this.destinationPath('public/app/routes/' + this._opts.route + '/index.js'),
      this._opts
    );
```

```
    this.fs.copyTpl(
      this.templatePath('template.html'),
      this.destinationPath('public/app/routes/' + this._opts.route +
        '/template.html'),
      this._opts
    );
```

```
  },
```

```
  'end': function () {
    this.log('Route `` + this._opts.route + `` created.');
```

```
  }
}
```

```
});
```

清单 3-14 所示的内容看起来和清单 3-9 所示的内容类似，主要的区别是 Yeoman 的 `NamedBase` 类的使用。通过 `NamedBase` 类的扩展来创建子脚手架，我们提示 Yeoman 该命令需要传入一个或者多个参数。清单 3-15 展示了新的脚手架 `route` 命令的使用。

清单 3-15 使用脚手架的 route 命令创建一个新的 Angular 路由

```
$ yo example:route users
  create public/app/routes/users/index.js
  create public/app/routes/users/template.html
Route `users` created.
```

3.3.4 可组合性

在创建 Yeoman 脚手架的时候，子命令间的互相调用是非常常见的情况。例如，我们刚创建一个脚手架，很容易出现这样一个场景。我们希望脚手架运行的时候可以自动创建多个默认的路由。为了实现这个目标，如果能在 app 命令中调用 route 命令，那将非常有用。Yeoman 的 `composeWith()` 方法就是因此存在的（见清单 3-16）。

清单 3-16 Yeoman 的 `composeWith()` 方法允许脚手架子命令间互相调用

```
// generator-example/app/index.js (excerpt)

'writing': function () {
  this.fs.copyTpl(
    this.templatePath('**/*'),
    this.destinationPath(),
    this._answers
  );

  this.fs.copy(
    this.templatePath('.bowerrc'),
    this.destinationPath('.bowerrc'),
    this._answers
  );

  /*
  Yeoman 的 'composeWith' 方法可以让我们执行外部的脚手架。此处，我们触发了新路由 “dashbord”
  的创建。
  */
  this.composeWith('example:route', {
    'args': ['dashboard']
  });

  this.config.save();
}
```

通过 Yeoman 的 `composeWith()` 方法，简单的脚手架子命令可以互相组合，以创建非常复杂的工作流。采用这个方法的优点在于，开发者可以创建复杂、多命令的脚手架，同时在不同命令间避免重复代码的使用。

3.4 小结

Yeoman 是一个简单但是功能强大的工具，通过将大量重复的任务自动化来引导一个新应用的创建，从而加速开发人员从概念到原型这个过程的处理。使用时，它允许开发者把注意力集中在最重要的地方——应用的本身。根据最新统计，已有超过 1 500 个 Yeoman 的脚手架在 npm 上发布，方便

开发人员尝试以前从未接触过的各种工具、库、框架、设计模式(如 Bower、Grunt、AngularJS、Knockout、React 等)。

3.5 相关资源

- Yeoman: <http://yeoman.io/>

PM2

光阴易逝，岂容我待，把握当下，立即行动，前进的路上会遇见更美的风景。

——乔治·赫伯特

第1部分中的前几章涵盖了各种有用的 Web 开发工具，我们的关注点都在客户端的开发上。在本章中，我们将提升开发工具的覆盖面，将焦点转移到服务端。我们将开始探索一个命令行的实用程序 PM2。它简化了许多与 Node 应用运行相关的任务，可以监视应用的运行状态，并且支持灵活的扩展以满足日益增长的需求。本章涵盖的主题包括：

- 理解进程
- 监控日志
- 监控资源使用情况
- 进程的高级管理
- 多处理器的负载均衡
- 0 秒宕机的部署方式

4.1 安装

PM2 的命令行工具 pm2 可以通过 npm 获取。如果您尚未安装 PM2，应该先按照清单 4-1 所列的命令进行操作。

清单 4-1 通过 npm 安装 pm2 的命令行工具

```
$ npm install -g pm2
$ pm2 --version
0.12.15
```

注意 用户通过 Node 的包管理器 (npm) 可以在两种环境下安装软件包：本地安装或者全局安装。在该示例中，我们将 bower 安装在了全局环境，这种方式通常用来安装命令行工具。

4.2 与进程一起工作

清单 4-2 展示的是一个简单的 Node 应用示例，我们将用该示例演示 PM2 的一些简单使用。当

运行该应用后，用户访问时会显示 “Hello, World”。

清单 4-2 一个简单的 Express 应用

```
// my-app/index.js

var express = require('express');
var morgan = require('morgan');
var app = express();
app.use(morgan('combined'));

app.get('/', function(req, res, next) {
  res.send('Hello, world.\n');
});

app.listen(8000);
```

图 4-1 演示了借助 pm2 命令行工具启动该应用的过程。在该示例中，PM2 通过执行我们指定的 index.js 脚本来启动应用。此外，我们还指定了一个（可选的）参数（my-app），以便我们后续引用。在此之前，请确保已经通过 `$ npm install` 安装了项目的依赖。

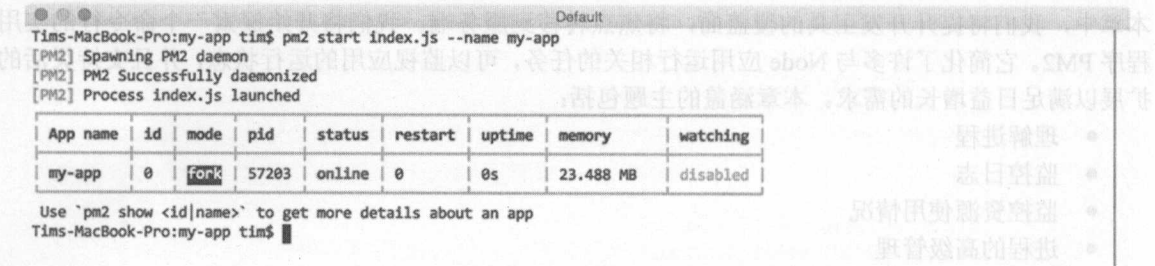


图 4-1 使用 PM2 启动清单 4-2 所示的应用

调用 PM2 的 start 命令后，PM2 会在返回命令提示符之前将当前所有 Node 应用相关的信息以表格的形式展示出来。其中每列的含义如表 4-1 所示。

表 4-1 图 4-1 中各列的含义汇总

表头	描述
App name	进程名称，默认为执行的脚本文件名
id	为 PM2 进程分配的一个唯一 ID，通过名称或 ID 均可以定位到具体进程
mode	进程的启动方式（fork 或 cluster），默认为 fork。稍后会在本章中详细介绍
pid	操作系统为进程分配的唯一数字，用于唯一标识每个进程
status	进程的当前状态（如 online、stopped 等）
restart	进程被 PM2 重启的次数
uptime	进程自上次重启后运行的时长
memory	进程所消耗的内存
watching	表明当项目的文件模式发生变化后，该进程是否会被 PM2 自动重启。在开发阶段特别有用，默认为 disabled

当运行 PM2 并看到如清单 4-3 所示的输出时，说明我们的应用已经在线并且可以被访问了。我们可以使用 curl 命令行工具访问应用的根路由来验证这一点，如图 4-2 所示。


```

Tims-MacBook-Pro:my-app tim$ curl http://localhost:8000
Hello, world.
Tims-MacBook-Pro:my-app tim$

```

图 4-2 访问我们 Express 应用的根路由

注意 图 4-2 中命令执行的前提是你已经安装 curl 命令行工具。如果不巧，当前工作环境中并未安装该工具，也可以通过 Web 浏览器直接访问来验证应用的运行状态。

除了 start 命令外，PM2 还提供了许多有用的命令用于操作当前已有的进程。最常见的几个命令如表 4-2 所示。

表 4-2 操作 PM2 进程的常用命令

命令	描述
list	显示如图 4-1 所示表格的最新状态
stop	停止进程，而不会从 PM2 的列表中移除
restart	重启进程
delete	停止并从 PM2 的列表中删除进程
show	显示特定进程的相关明细

像 stop、start 和 delete 这样简单的命令无须指定更多参数。相反，如图 4-3 所示，通过 show 命令可以获取某个特定 PM2 进程的详细信息。

```

Tims-MacBook-Pro:my-app tim$ pm2 show my-app
Describing process with id 0 - name my-app

```

status	online
name	my-app
id	0
path	/Users/tim/repos/pro-javascript-frameworks/code/pm2/my-app/index.js
args	
exec cwd	/Users/tim/repos/pro-javascript-frameworks/code/pm2/my-app
error log path	/Users/tim/.pm2/logs/my-app-error-0.log
out log path	/Users/tim/.pm2/logs/my-app-out-0.log
pid path	/Users/tim/.pm2/pids/my-app-0.pid
mode	fork_mode
node v8 arguments	
watch & reload	x
interpreter	node
restarts	0
unstable restarts	0
uptime	5m
created at	2015-07-27T13:02:22.410Z

```

Revision control metadata

```

revision control	git
remote url	git@github.com:tkambler/pro-javascript-frameworks.git
repository root	/Users/tim/repos/pro-javascript-frameworks
last update	2015-07-27T13:07:22.000Z
revision	5b437670e51606d68e1184051ebba8e194a0034
comment	Replaces absolute path with relative path
branch	pm2

```

Tims-MacBook-Pro:my-app tim$

```

图 4-3 查看某个特定 PM2 进程的详细信息

4.2.1 从错误中恢复

至此，你已经熟悉了 PM2 的一些基础使用步骤。你已经了解如何使用 PM2 的 `start` 命令创建一个新的进程，也接触了如果使用诸如 `list`、`stop`、`restart`、`delete` 和 `show` 命令对运行中的进程进行后续管理。

虽然我们还没有展开讨论，不过实际上表中的每个值都和 Node 的进程管理有关，PM2 会帮助 Node 应用从致命的错误中自动恢复，让我们从这里说起。

清单 4-3 所示代码是我们之前在清单 4-2 中所见应用的修改版本。不同的是，在该版本中会定期抛出一个异常。

清单 4-3 原有应用的修改版本，将会每隔 4 秒抛出一个异常

```
// my-bad-app/index.js
```

```
var express = require('express');
var morgan = require('morgan');
var app = express();
app.use(morgan('combined'));

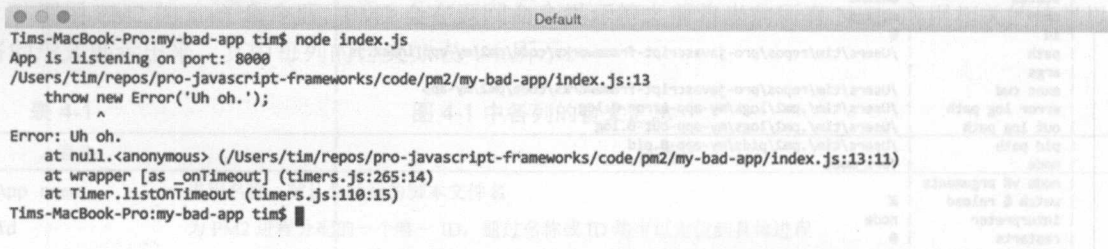
app.get('/', function(req, res, next) {
  res.send('Hello, world.\n');
});
```

```
setInterval(function() {
  throw new Error('Uh oh.');
```

```
}, 4000);

app.listen(8000);
```

倘若我们不使用 PM2 而直接使用 Node 运行该应用的话，将会很快不幸地看到所抛出的第一个错误。Node 只是简单地将错误信息打印在控制台，之后便返回命令提示符，如图 4-4 所示。



```

Tims-MacBook-Pro:my-bad-app tim$ node index.js
App is listening on port: 8000
/Users/tim/repos/pro-javascript-frameworks/code/pm2/my-bad-app/index.js:13
  throw new Error('Uh oh.');
```

^

```

Error: Uh oh.
    at null.<anonymous> (/Users/tim/repos/pro-javascript-frameworks/code/pm2/my-bad-app/index.js:13:11)
    at wrapper [as _onTimeout] (timers.js:265:14)
    at Timer.listOnTimeout (timers.js:110:15)
Tims-MacBook-Pro:my-bad-app tim$
```

图 4-4 当 Node 执行清单 4-3 中的代码报错崩溃后输出的信息

这种情况在真实的使用场景中离我们并不遥远。理想情况下，被发布到生产环境的应用应当是经过全面测试并且不该存在这类未捕获异常的。不过，当应用崩溃后至少应该能够自动恢复而无需人工干预。PM2 就可以帮助我们实现这个目标。

图 4-5 中，我们通过 `delete` 命令从 PM2 的列表中删除了现存的进程，并为清单 4-3 所示糟糕的应用代码创建一个新的实例。之后等待几秒，然后查看最新的 PM2 的进程列表。

有没有注意到什么有趣的现象？根据 `status`、`restart` 和 `uptime` 几栏的值，可见我们的应用已经崩溃了三次。每次崩溃时，PM2 都会帮我们重启应用。自最近一次重启后，进程共计已经运行了两秒。也就是说，我们可以想象两秒后会有另外一次崩溃（并且自动重启）。

```

Tims-MacBook-Pro:my-bad-app tim$ pm2 delete my-app
[PM2] Deleting my-app process
[PM2] deleteProcessId process id 0

```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-bad-app	1	fork	57530	online	0	0s	22.344 MB	disabled

```

Use `pm2 show <id|name>` to get more details about an app
Tims-MacBook-Pro:my-bad-app tim$ pm2 start index.js --name my-bad-app
[PM2] Process index.js launched

```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-bad-app	1	fork	57561	online	2	3s	30.301 MB	disabled

```

Use `pm2 show <id|name>` to get more details about an app
Tims-MacBook-Pro:my-bad-app tim$

```

图 4-5 PM2 帮助 Node 应用从致命的错误中恢复

PM2 能够帮助生产环境的应用从致命的错误中恢复，这点固然有用，但相比其提供的众多特性而言却只是其中之一。接下来我们会发现 PM2 其实在开发环境中同样有用。

4.2.2 监控文件变化

设想这样一个场景：假设你在一个全新的 Node 项目中使用 Express 开发一个 Web API。倘若不借助任何辅助工具的话，当你进行一些改动后必须得手动重启相关的 Node 进程来查看效果——这是一件消磨时光且令人沮丧的苦差事。针对这种情况，PM2 可以自动帮你监控项目的文件模式。一旦发生变化，PM2 会帮你自动重启应用。

图 4-6 演示了这个过程。在本例中，我们首先删除 my-bad-app 应用当前正在运行的实例。接下来，我们为前面例子中的应用创建一个新的实例（见清单 4-2）。但这次我们通过一个额外的参数 --watch 指示 PM2 监控我们的项目变化并自动做出响应。

```

Tims-MacBook-Pro:my-app tim$ pm2 delete my-bad-app
[PM2] Deleting my-bad-app process
[PM2] deleteProcessId process id 1

```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-app	2	fork	58021	online	0	0s	22.004 MB	enabled

```

Use `pm2 show <id|name>` to get more details about an app
Tims-MacBook-Pro:my-app tim$ pm2 start index.js --name my-app --watch
[PM2] Process index.js launched

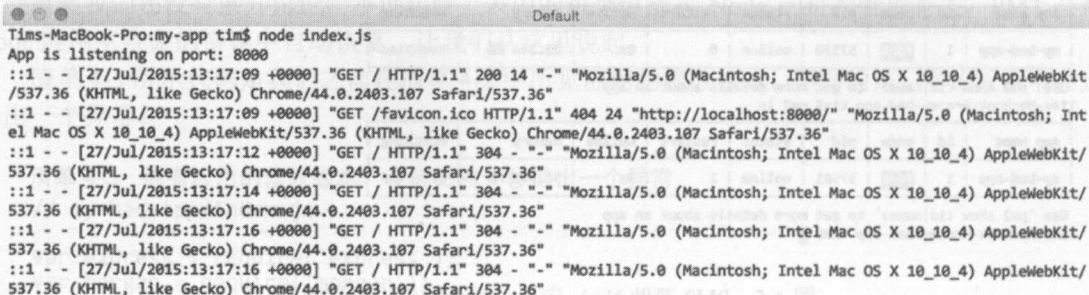
```

图 4-6 创建一个新的 PM2 进程，当文件发生变化时会自动重启

当该项目中的文件发生变化并保存后，将像前面的例子一样，调用 PM2 的 list 命令，显示出 PM2 重启应用的次数。

4.3 监控日志

回顾一下清单 4-2，你会发现该应用中用到了 `morgan`。它是一个用于记录 HTTP 请求的模块。在这个例子中，`morgan` 被配置为将这些信息打印在控制台。直接通过 Node 运行我们的应用即可看到这些信息，如图 4-7 所示。



```

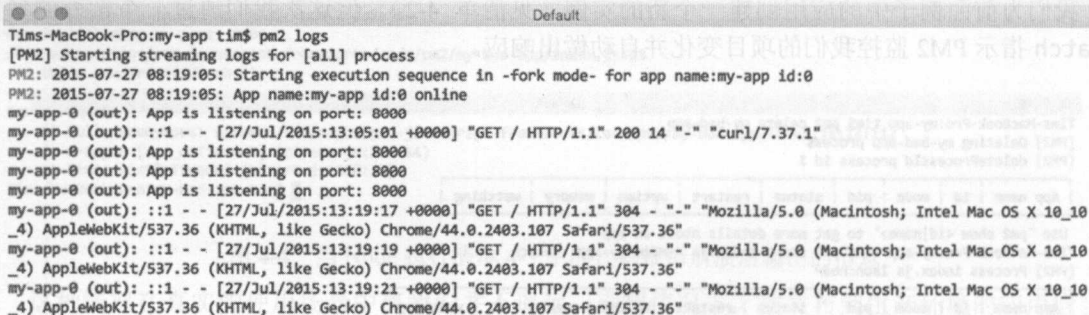
Tims-MacBook-Pro:my-app tim$ node index.js
App is listening on port: 8000
::1 - - [27/Jul/2015:13:17:09 +0000] "GET / HTTP/1.1" 200 14 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
::1 - - [27/Jul/2015:13:17:09 +0000] "GET /favicon.ico HTTP/1.1" 404 24 "http://localhost:8000/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
::1 - - [27/Jul/2015:13:17:12 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
::1 - - [27/Jul/2015:13:17:14 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
::1 - - [27/Jul/2015:13:17:16 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
::1 - - [27/Jul/2015:13:17:16 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"

```

图 4-7 使用 `morgan` 记录 Express 的请求

我们之前探讨了如何使用 PM2 的 `start` 命令来启动该应用（见图 4-1）。有得有失，这样做虽然为我们提供了一些好处，但却无法在控制台实时看到应用产生的输出。幸运的是，PM2 为我们提供了一个简单的机制来监控这类输出。

在图 4-3 中，我们通过 `show` 命令查看某个特定 PM2 进程的详细信息。提供的信息中有 PM2 为该进程生成的两个日志文件的路径——一个是“`out log path`”，另一个是“`error log path`”——PM2 将把进程的标准输出和错误信息分别保存在这两个文件中。我们可以直接查看这些文件，不过 PM2 提供了更方便的方法来查看这些日志，如图 4-8 所示。



```

Tims-MacBook-Pro:my-app tim$ pm2 logs
[PM2] Starting streaming logs for [all] process
PM2: 2015-07-27 08:19:05: Starting execution sequence in -fork mode- for app name:my-app id:0
PM2: 2015-07-27 08:19:05: App name:my-app id:0 online
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): ::1 - - [27/Jul/2015:13:19:17 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
my-app-0 (out): ::1 - - [27/Jul/2015:13:19:19 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
my-app-0 (out): ::1 - - [27/Jul/2015:13:19:21 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"

```

图 4-8 监控在 PM2 控制之下的进程的输

可见通过 PM2 的 `logs` 命令，我们可以如期监控到进程的输。在这个例子中，我们监控了所有在 PM2 控制之内的进程输出。值得注意的是，PM2 为每一条内容都添加了前缀，能够直观地看出该条内容是哪个进程所产生的。当使用 PM2 管理多个进程时，这些信息非常有用。下一节中将介绍这部分内容。另外，我们还可以在使用 `logs` 命令时通过指定进程的名称（或者 ID）来监控某个特定进

程的输出（见图 4-9）。

```

Tims-MacBook-Pro:my-app tim$ pm2 logs my-app
[PM2] Starting streaming logs for [my-app] process
PM2: 2015-07-27 08:19:05: Starting execution sequence in -fork mode- for app name:my-app id:0
PM2: 2015-07-27 08:19:05: App name:my-app id:0 online
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): ::1 - - [27/Jul/2015:13:05:01 +0000] "GET / HTTP/1.1" 200 14 "-" "curl/7.37.1"
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): App is listening on port: 8000
my-app-0 (out): ::1 - - [27/Jul/2015:13:19:17 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
my-app-0 (out): ::1 - - [27/Jul/2015:13:19:19 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"
my-app-0 (out): ::1 - - [27/Jul/2015:13:19:21 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.107 Safari/537.36"

```

图 4-9 通过 PM2 的控制监控某个特定进程的输

如果你想清除 PM2 生成的日志文件内容，可以随时调用 PM2 的 flush 命令来完成。logs 命令可以接受两个可选的参数，如表 4-3 所示。

表 4-3

PM2 的 logs 命令可以接受的参数

参数	描述
-raw	显示日志文件的原始内容，不包含标识进程的前缀
-lines	只显示最后 N 行，默认显示 20 行

4.4 监控资源占用

上一节中介绍了如何使用 PM2 帮你监控进程所产生的标准输出和错误信息。针对进程以及整个服务器的运行状况，PM2 同样提供了易于使用的工具。

4.4.1 监控本地资源

图 4-10 展示了调用 PM2 的 monit 命令时产生的输出。我们可以通过实时更新的界面查看 PM2 管理的每个进程的 CPU 的占用以及内存消耗情况。

```

PM2 monitoring :
• my-app
[0] [fork_mode]
[ ] 0 %
[ ] 30.953 MB

```

图 4-10 通过 PM2 的 monit 命令监控 CPU 和内存的使用情况

4.4.2 监控远程资源

通过 PM2 的 monit 命令所提供的信息，我们可以方便快捷地监控进程的运行状况。在开发阶段，这个功能非常有用，可以令我们对自己环境的资源消耗情况了如指掌。不过一旦应用被部署到远程的生产环境，就可能涉及多台服务器，每台服务器都会运行自己的 PM2 实例，这时候 monit 命令就显得有些捉襟见肘了。

PM2 针对这种情况提供了一个内置的 JSON API 接口，可以通过 9615 端口访问。该功能默认被禁用，启用它的步骤如图 4-11 所示。

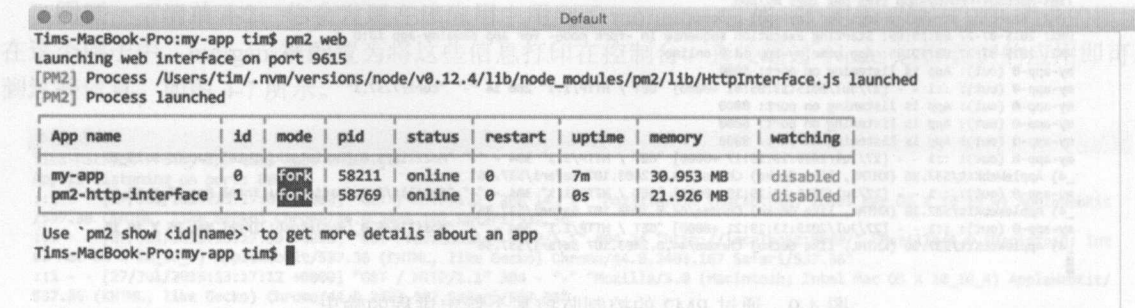


图 4-11 启用 PM2 的 JSON Web API 接口

在这个例子中，我们通过调用 Web 命令来启用 PM2 的网络访问 JSON API 接口。PM2 将该功能作为一个独立于自身的应用单独运行。此时你会注意到，在 PM2 的控制下出现一个新的进程 pm2-http-interface。当你希望禁用 PM2 的 JSON API 时，只需要像移除其他进程一样，将其名称（或者 ID）传递给 delete（或者 stop）命令。

向运行在端口 9615 的服务器发送 GET 请求后会得到该 API 返回的内容，清单 4-4 所示是输出内容的一部分。正如你所见，PM2 为我们提供了它管理的每个进程以及 PM2 系统相关的一些详细信息。

清单 4-4 PM2 的 JSON API 接口返回的一部分信息

```
{
  "system_info": {
    "hostname": "iMac.local",
    "uptime": 2186
  },
  "monit": {
    "loadavg": [1.39794921875],
    "total_mem": 8589934592,
    "free_mem": 2832281600,
    "cpu": [{
      "model": "Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz",
      "speed": 3300,
      "times": {
        "user": 121680,
        "nice": 0,
        "sys": 176220,
        "idle": 1888430,
        "irq": 0
      }
    }
  ],
  "interfaces": {
    "lo0": [{
      "address": ":::1",
      "netmask": "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff",
      "family": "IPv6",
      "mac": "00:00:00:00:00:00",
      "scopeid": 0,
      "internal": true
    }
  ],
  "en0": [{
```

```

    "address": "10.0.1.49",
    "netmask": "255.255.255.0",
    "family": "IPv4",
    "mac": "ac:87:a3:35:9c:72",
    "internal": false
  }
},
"processes": [{
  "pid": 1163,
  "name": "my-app",
  "pm2_env": {
    "name": "my-app",
    "vizion": true,
    "autorestart": true,
    "exec_mode": "fork_mode",
    "exec_interpreter": "node",
    "pm_exec_path": "/opt/my-app/index.js",
    "env": {
      "": "/usr/local/opt/nvm/versions/node/v0.12.4/bin/pm2",
      "NVM_IOJS_ORG_MIRROR": "https://iojs.org/dist",
      "NVM_BIN": "/usr/local/opt/nvm/versions/node/v0.12.4/bin",
      "LOGNAME": "user",
      "ITERM_SESSION_ID": "w0t0p0",
      "HOME": "/Users/user",
      "COLORFGBG": "7;0",
      "SHLVL": "1",
      "XPC_SERVICE_NAME": "0",
      "XPC_FLAGS": "0x0",
      "ITERM_PROFILE": "Default",
      "LANG": "en_US.UTF-8",
      "PWD": "/opt/my-app",
      "NVM_NODEJS_ORG_MIRROR": "https://nodejs.org/dist",
      "PATH": "/usr/local/opt/nvm/versions/node/v0.12.4/bin",
      "CF_USER_TEXT_ENCODING": "0x1F5:0x0:0x0",
      "SSH_AUTH_SOCK": "/private/tmp/com.apple.launchd.kEqu8iouDS/Listeners",
      "USER": "user",
      "NVM_DIR": "/usr/local/opt/nvm",
      "NVM_PATH": "/usr/local/opt/nvm/versions/node/v0.12.4/lib/node",
      "TMPDIR": "/var/folders/y3/2fphz1fd6rg9l4cg2t8t7g840000gn/T",
      "TERM": "xterm",
      "SHELL": "/bin/bash",
      "TERM_PROGRAM": "iTerm.app",
      "NVM_IOJS_ORG_VERSION_LISTING": "https://iojs.org/dist/index.tab",
      "pm_cwd": "/opt/my-app"
    }
  },
  "versioning": {
    "type": "git",
    "url": "git@github.com:tkambler/pro-javascript-frameworks.git",
    "revision": "18104d13d14673652ee7a522095fc06dcf87f8ba",
    "update_time": "2015-05-25T20:53:50.000Z",
    "comment": "Merge pull request #28 from tkambler/ordered-build",
    "unstaged": true,
    "branch": "pm2",
    "remotes": ["origin"],
    "remote": "origin",
    "branch_exists_on_remote": false,
    "ahead": false,
    "next_rev": null,
    "prev_rev": "b0e486adab79821d3093c6522eb8a24455bfb051",
    "repo_path": "/Users/user/repos/pro-javascript-frameworks"
  }
}]
}

```

```
    },
    "pm_id": 0,
    "monit": {
      "memory": 32141312,
      "cpu": 0
    }
  }
}
```

4.5 进程的高级管理

本章的大部分重点主要围绕着如何通过命令行使用 PM2。start、stop、restart 和 delete 命令为我们提供了一种高效管理进程的简单机制，但对于更复杂的场景呢？或许一个应用在启动的时候需要额外的参数，或是需要进行一个或者多个环境变量的设置。

通过 JSON 方式配置应用

为了满足这些需求，我们需要进行额外的配置。好在 PM2 支持通过 JSON 方式配置应用，这也是实现这一目标的最佳途径。清单 4-5 是一个配置文件的示例，演示了大多数可用的配置项。

清单 4-5 一个包含大多数可用选项的 JSON 配置文件示例

```
{
  "name"      : "my-app",
  "cwd"       : "/opt/my-app",
  "args"      : ["--argument1=value", "--flag", "value"],
  "script"    : "index.js",
  "node_args" : ["--harmony"],
  "log_date_format" : "YYYY-MM-DD HH:mm Z",
  "error_file" : "/var/log/my-app/err.log",
  "out_file"   : "/var/log/my-app/out.log",
  "pid_file"   : "pids/my-app.pid",
  "instances"  : 1, // or 0 => 'max'
  "max_restarts" : 10, // defaults to 15
  "max_memory_restart" : "1M", // 1 megabytes, e.g.: "2G", "10M", "100K"
  "cron_restart" : "1 0 * * *",
  "watch"      : false,
  "ignore_watch" : ["node_modules"],
  "merge_logs"  : true,
  "exec_mode"   : "fork",
  "autorestart" : false,
  "env"         : {
    "NODE_ENV" : "production"
  }
}
```

JSON 应用配置文件通过标准的格式进行 PM2 的高级设置，为我们提供了一种易于复用和共享的方式。或许你会觉得一些选项很眼熟，比如前面示例中提到的这些（诸如 name、out_file、error_file、watch 等）。其他选项在本章后面部分会提到。每个配置项的具体描述如表 4-4 所示。

表 4-4 清单 4-5 中所涉及众多配置项的描述

参数	描述
name	应用的名称
cwd	被启动应用所在的目录

续表

参数	描述
Args	传递给应用的命令行参数
script	应用启动的脚本路径（相对于 cwd）
node_args	传递给 node 的命令行参数
log_date_format	日志时间戳的生成格式
error_file	标准错误输出的日志存储路径
out_file	标准输出的日志存储路径
pid_file	应用的 PID（进程标示符）日志文件路径
instances	应用启动的实例数，在下一节中将进一步介绍
max_restarts	当应用崩溃后，PM2 尝试（连续）重启的最大次数
max_memory_restart	当内存占用超过设定的阈值时，PM2 将自动重启应用
cron_restart	PM2 将按照设定的计划自动重启应用
watch	当 PM2 检测到文件模式发生变化时是否自动重启应用，默认为 false
ignore_watch	当启用文件监控后，数组中包含的指定路径将被忽略
merge_logs	如果一个应用启动多个实例，PM2 将分别合并所有实例的标准输出和错误输出到一个文件中
exec_mode	运行的方式，默认为 fork，在下一节中将会进一步介绍
autorestart	当应用崩溃或者异常退出时是否自动重启，默认为 true
vizion	当启用后，PM2 将尝试读取可能存在的版本控制文件的元数据，默认为 true
env	对象中以 keys/values 形式包含的环境变量将被传递给应用

本章所包含的是一个微服务（microservices）项目，将演示 JSON 配置文件的实际使用。项目中包含两个应用：一个天气应用（weather）和一个主应用（main）。天气应用提供的 API 将针对特定的邮编返回随机的温度信息，主应用每隔两秒钟会向 API 发送请求并将结果输出到控制台。每个应用的主要脚本代码如清单 4-6 所示。

清单 4-6 主应用（main）和天气应用（weather）的源代码

```
// microservices/main/index.js

var request = require('request');

if (!process.env.WEATHER_API_URL) {
  throw new Error('The `WEATHER_API_URL` environment variable must be set.');
```

```
}

setInterval(function() {
  request({
    'url': process.env.WEATHER_API_URL + '/api/weather/37204',
    'json': true,
    'method': 'GET'
  }, function(err, res, result) {
    if (err) throw new Error(err);
    console.log('The temperature is: %s', result.temperature.fahrenheit);
  });
}, 2000);
```

```
// microservices/weather/index.js

if (!process.env.PORT) {
  throw new Error('The `PORT` environment variable must be set.');
```

```
}

var express = require('express');
var morgan = require('morgan');
var app = express();
app.use(morgan('combined'));

var random = function(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
};

app.get('/api/weather/:postal_code', function(req, res, next) {
  var fahr = random(70, 110);
  res.send({
    'temperature': {
      'fahrenheit': fahr,
      'celsius': (fahr - 32) * (5/9)
    }
  });
});

app.listen(process.env.PORT);
```

微服务项目中还包含单独的 JSON 应用配置文件，其内容如清单 4-7 所示。

清单 4-7 本章中微服务项目(microservices)的 JSON 应用配置文件 microservices/pm2/development.json

```
[
  {
    "name"           : "main",
    "cwd"            : "../microservices",
    "script"         : "main/index.js",
    "max_memory_restart" : "60M",
    "watch"          : true,
    "env"            : {
      "NODE_ENV" : "development",
      "WEATHER_API_URL": "http://localhost:7010"
    }
  },
  {
    "name"           : "weather-api",
    "cwd"            : "../microservices",
    "script"         : "weather/index.js",
    "max_memory_restart" : "60M",
    "watch"          : true,
    "env"            : {
      "NODE_ENV" : "development",
      "PORT"      : 7010
    }
  }
]
```

如上所示的 PM2 应用配置文件定义了每个应用的启动方式。在这个例子中，当 PM2 检测到文件模式发生变化，或者内存占用超过 60 MB 时，都将自动重启每个应用。同时，还为每个进程单独配置了环境变量。

■ **注意** 运行该示例之前，首先要调整 `cwd` 的设置，以确保对应的绝对路径能够引用到 `microservices` 目录。之后通过 PM2 即可一次性启动两个应用，如图 4-12 所示。

```
Tims-MacBook-Pro:microservices tim$ pm2 start ./pm2/development.json
[PM2] Spawning PM2 daemon
[PM2] PM2 Successfully daemonized
[PM2] Process launched
[PM2] Process launched
```

App name	id	mode	pid	status	restart	uptime	memory	watching
main	0	fork	58873	online	0	0s	27.242 MB	enabled
weather-api	1	fork	58874	online	0	0s	20.875 MB	enabled

```
Use `pm2 show <id|name>` to get more details about an app
Tims-MacBook-Pro:microservices tim$
```

图 4-12 使用 PM2 同时启动 main 和 weather-api 两个应用

如我们所预期，PM2 将会为配置文件中描述的两个应用分别创建两个实例。同前面的例子一样，我们可以通过 PM2 的 `logs` 命令监控所产生的日志输出（见图 4-13）。

```
Tims-MacBook-Pro:microservices tim$ pm2 logs
[PM2] Starting streaming logs for [all] process
main-0 (out): The temperature is: 84
weather-api-1 (out): ::ffff:127.0.0.1 - - [27/Jul/2015:13:29:15 +0000] "GET /api/weather/37204 HTTP/1.1" 200 61 "-" "-"
weather-api-1 (out): ::ffff:127.0.0.1 - - [27/Jul/2015:13:29:17 +0000] "GET /api/weather/37204 HTTP/1.1" 200 46 "-" "-"
main-0 (out): The temperature is: 77
weather-api-1 (out): ::ffff:127.0.0.1 - - [27/Jul/2015:13:29:19 +0000] "GET /api/weather/37204 HTTP/1.1" 200 61 "-" "-"
main-0 (out): The temperature is: 73
weather-api-1 (out): ::ffff:127.0.0.1 - - [27/Jul/2015:13:29:21 +0000] "GET /api/weather/37204 HTTP/1.1" 200 62 "-" "-"
main-0 (out): The temperature is: 78
weather-api-1 (out): ::ffff:127.0.0.1 - - [27/Jul/2015:13:29:23 +0000] "GET /api/weather/37204 HTTP/1.1" 200 61 "-" "-"
main-0 (out): The temperature is: 70
weather-api-1 (out): ::ffff:127.0.0.1 - - [27/Jul/2015:13:29:25 +0000] "GET /api/weather/37204 HTTP/1.1" 200 61 "-" "-"
main-0 (out): The temperature is: 91
```

图 4-13 PM2 的 `logs` 命令所输出的内容片段

4.6 多核处理器的负载均衡

得益于 Node 的单线程、非阻塞 I/O 模型特性，开发人员可以更容易地创建能够处理成千上万高并发的应用。但与此同时带来的弊端是无法充分利用多核 CPU 的资源。不过值得庆幸的是，使用 Node 的核心模块 `cluster` 可以突破这一限制。通过 `cluster` 模块，开发者在应用中能够自己创建子进程——每个进程都运行在不同的处理器上，并且能够同其他子进程和创建它的父进程共享端口。

在本章结束之前，让我们先来了解一下 PM2 针对 Node 的 `cluster` 模块所提供的便捷功能。通过这个功能，即便应用原本没有使用 Node 的 `cluster` 模块，也可以通过某种方式启动并充分利用多核处理器的资源。这样一来，针对日益增长的需求，开发者就可以快速扩展他们的应用，而无须再迫不得已通过增加额外的服务器来解决了。

清单 4-8 所示代码是一个简单 Express 应用的源码。我们将借助 PM2 将应用扩展到多个处理器，对应的 JSON 应用配置文件如清单 4-9 所示。

清单 4-8 将 Express 应用扩展为使用多核 CPU

```
// multicore/index.js

if (!process.env.port) throw new Error('The port environment variable must be set');
```

```
var express = require('express');
var morgan = require('morgan');
var app = express();
app.use(morgan('combined'));

app.route('/')
  .get(function(req, res, next) {
    res.send('Hello, world.');
```

清单 4-9 启动应用所需的 JSON 配置文件
// multicore/pm2/development.json

```
{
  "name": "multicore",
  "cwd": "../multicore",
  "max_memory_restart": "60M",
  "watch": false,
  "script": "index.js",
  "instances": 0, // max
  "exec_mode": "cluster",
  "autorestart": true,
  "merge_logs": true,
  "env": {
    "port": 9000
  }
}
```

清单 4-9 所示的应用配置文件中有两个比较关键的设置。第一个是 `instances` 属性。在这个例子中，我们设置的值为 0，此时 PM2 会按照 CPU 的个数启动不同的独立进程。另一个是 `exec_mode` 属性。当设置为 `cluster` 时，PM2 会启动自己的父进程，父进程通过 Node 的 `cluster` 模块反过来会为应用启动独立的进程。

如图 4-14 所示，我们将应用的配置文件作为参数传递给 PM2 的 `start` 命令，以此启动应用。随后，PM2 会像前面的例子一样列出所有的已知进程。此时会发现，PM2 为 8 个 CPU 分别启动了不同的独立进程。我们可以执行 `monit` 命令，通过监控这些新进程的 CPU 占用来验证这一点，如图 4-15 所示。

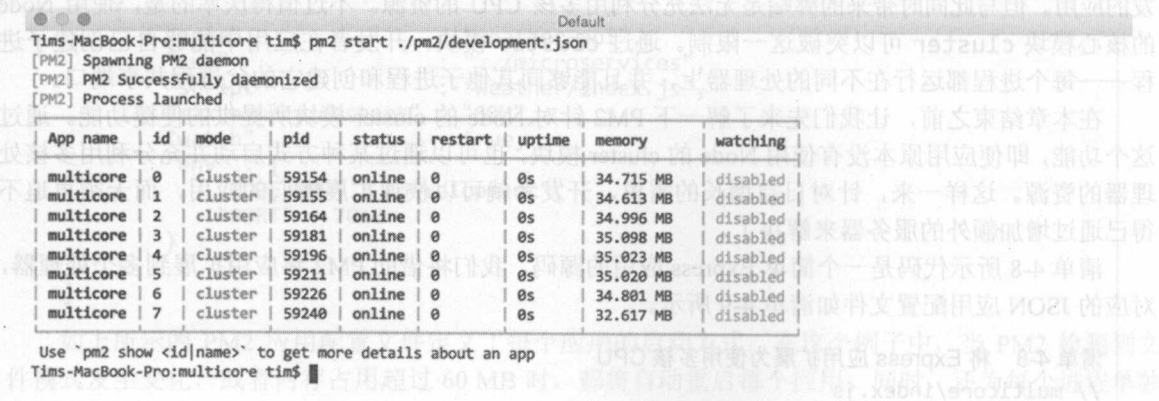


图 4-14 通过 PM2 的 cluster 模式启动应用

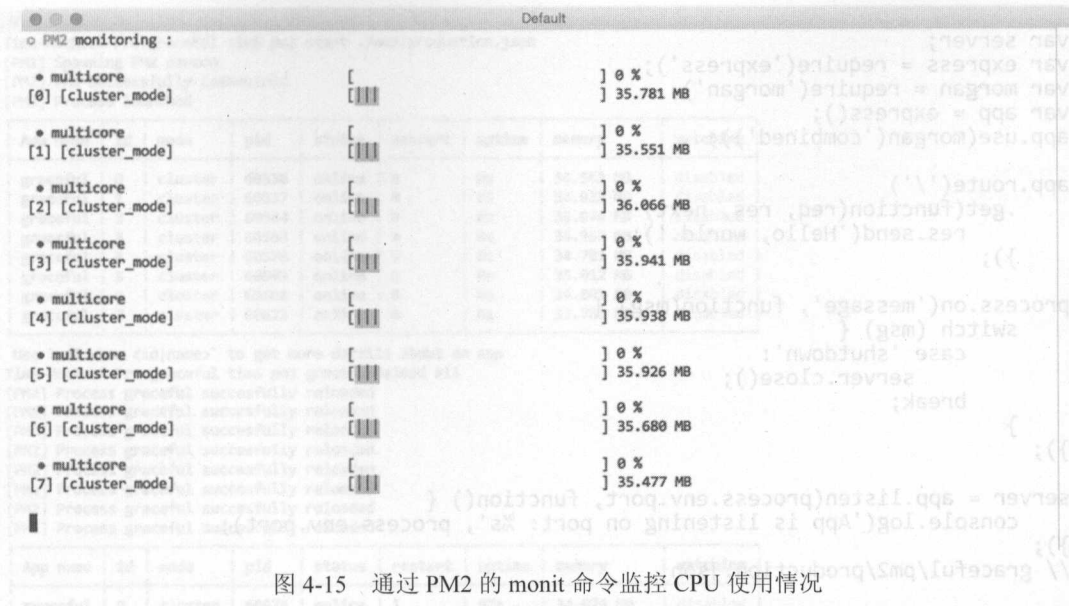


图 4-15 通过 PM2 的 monit 命令监控 CPU 使用情况

注意 当使用 cluster 模式启动应用时,PM2 会在控制台警告用户该功能尚处在测试阶段。据 PM2 的核心开发人员透露,只要所用的 Node 版本不低于 v0.12.0,该功能就足够稳定且可以应用到生产环境。

在继续后面的内容之前,可以先执行 `$ pm2 delete multicore` 快速移除该示例启动的 8 个进程。

0 秒宕机的部署方式

通过 cluster 模式启动应用后,PM2 将通过轮询调度 (round-robin) 的方式依次将请求转发给 8 个不同的处理器,从而带来巨大的性能提升。将我们的应用分布在多个处理器上还有另一个好处,即当我们发布更新时不会产生任何的宕机时间,接下来会详细解释。

设想如下场景:一个通过 PM2 进行管理的应用运行在一台或者多台服务器上。当有更新需要发布时涉及两个关键步骤:

- 将更新后的源代码复制到相应的服务器上
- 通过 PM2 重启每个进程

正是这样的步骤,将会导致短暂的服务宕机,在此期间访问应用的请求都将被拒绝,除非采取特殊措施。幸运的是,通过 PM2 以 cluster 模式启动应用时,我们可以借助其提供的工具避免这样的情况发生。

以先前清单 4-8 中的应用为例,如果在重启应用时需要避免任何时间的宕机,我们首先需要对应用的源码和应用的配置文件稍做修改。更新之后的版本如清单 4-10 所示。

清单 4-10 修改之后的应用可以利用 PM2 的 gracefulReload 命令避免重启时宕机

```
// graceful/index.js
if (!process.env.port) throw new Error('The port environment variable must be set');
```

```

var server;
var express = require('express');
var morgan = require('morgan');
var app = express();
app.use(morgan('combined'));

app.route('/')
  .get(function(req, res, next) {
    res.send('Hello, world.');
```

图 4-12 通过 PM2 的 `start` 命令启动应用

```

  });

process.on('message', function(msg) {
  switch (msg) {
    case 'shutdown':
      server.close();
      break;
  }
});

server = app.listen(process.env.port, function() {
  console.log('App is listening on port: %s', process.env.port);
});
// graceful/pm2/production.json
{
  "name": "graceful",
  "cwd": "../graceful",
  "max_memory_restart": "60M",
  "watch": false,
  "script": "index.js",
  "instances": 0, // max
  "exec_mode": "cluster",
  "autorestart": true,
  "merge_logs": false,
  "env": {
    "port": 9000,
    "PM2_GRACEFUL_TIMEOUT": 10000
  }
}
```

图 4-13 通过 PM2 的 `start` 命令启动应用

前面的例子中已经演示了 PM2 的 `restart` 命令，执行后将停止并启动特定的进程。在非生产环境中使用并无大碍，然而考虑到命令执行时依旧可能会有活动的请求正在处理，此时若继续使用该命令就会有负面影响。当视稳定性为第一要义时，PM2 的 `gracefulReload` 命令是一个更为合适的选择。

当调用 `gracefulReload` 命令后，PM2 首先会向所管理的每个进程发送 `shutdown` 消息，以便这些进程有机会采取必要的措施来确保活动连接不会受到干扰。只有过了所配置的时间之后（通过 `PM2_GRACEFUL_TIMEOUT` 环境变量指定），PM2 才会对进程执行重启操作。

在这个例子中，当应用接收到 `shutdown` 消息后，会调用 `close()` 方法关闭 Express 所创建的 HTTP 服务。之后我们的服务将停止接受新的连接，但那些已经建立的连接不受影响。只有在 10 秒后（通过 `PM2_GRACEFUL_TIMEOUT` 指定的时间），PM2 才会重启进程，而此时该进程中那些已有的连接应该都已经处理完了。

图 4-16 演示了应用启动之后通过 `gracefulReload` 命令重启进程。这样，我们就能够在不打扰用户的前提下发布应用的更新了。

```

Tims-MacBook-Pro:graceful tim$ pm2 start ./pm2/production.json
[PM2] Spawning PM2 daemon
[PM2] PM2 Successfully daemonized
[PM2] Process launched

```

App name	id	mode	pid	status	restart	uptime	memory	watching
graceful	0	cluster	60536	online	0	0s	34.563 MB	disabled
graceful	1	cluster	60537	online	0	0s	34.922 MB	disabled
graceful	2	cluster	60544	online	0	0s	35.074 MB	disabled
graceful	3	cluster	60563	online	0	0s	34.953 MB	disabled
graceful	4	cluster	60578	online	0	0s	34.781 MB	disabled
graceful	5	cluster	60593	online	0	0s	35.012 MB	disabled
graceful	6	cluster	60608	online	0	0s	34.801 MB	disabled
graceful	7	cluster	60623	online	0	0s	32.781 MB	disabled

```

Use `pm2 show <id|names>` to get more details about an app
Tims-MacBook-Pro:graceful tim$ pm2 gracefulReload all
[PM2] Process graceful successfully reloaded
[PM2] Process graceful successfully reloaded
[PM2] Process graceful successfully reloaded
[PM2] Process graceful successfully reloaded
[PM2] Process graceful successfully reloaded
[PM2] Process graceful successfully reloaded
[PM2] Process graceful successfully reloaded

```

App name	id	mode	pid	status	restart	uptime	memory	watching
graceful	0	cluster	60674	online	1	67s	34.078 MB	disabled
graceful	1	cluster	60694	online	1	58s	34.633 MB	disabled
graceful	2	cluster	60713	online	1	50s	34.430 MB	disabled
graceful	3	cluster	60865	online	1	42s	34.652 MB	disabled
graceful	4	cluster	60887	online	1	33s	34.520 MB	disabled
graceful	5	cluster	60906	online	1	25s	34.602 MB	disabled
graceful	6	cluster	60925	online	1	16s	34.711 MB	disabled
graceful	7	cluster	61078	online	1	8s	34.668 MB	disabled

```

Use `pm2 show <id|names>` to get more details about an app
Tims-MacBook-Pro:graceful tim$

```

图 4-16 通过 PM2 的 gracefulReload 命令优雅地重启各个进程

4.7 小结

无论生产环境还是非生产环境，PM2 都是一个强大的 Node 应用管理工具。它包含了一些简单的功能，诸如源码发生改变后自动重启进程，以便节省开发时间；也包含一些高级特性，比如支持多核处理器的负载均衡，可以在不影响用户的情况下优雅地重启应用，还为更好地使用 Node 提供了一些其他重要的功能。

4.8 相关资源

- PM2: <https://github.com/Unitech/pm2>

RequireJS

更卓有成效的思考是思考什么是在我的控制之中，而不是为我无法控制的事情担心和发愁。

——皮特·圣安德烈

虽然 JavaScript 现在在 Web 应用程序中扮演了更重要的角色，HTML 5 规范（现代浏览器）却没有一种方法来检测脚本之间的依赖关系，也无法按照特定顺序加载脚本和依赖。在最简单的场景中，脚本通常是指在页面中引用的<script>标签。这些标签被解析、加载，并按顺序执行，这意味着公共库或者模块一般要放在第一位，然后是应用本身的脚本。（例如，一个页面可能是先加载 jQuery，然后加载应用本身的脚本，这些脚本使用 jQuery 来操作文档对象模型[DOM]。）简单的页面依赖关系也比较简单，使用上述方法尚且够用，但是随着 Web 应用复杂性的增加，应用脚本的数量也随之增加，互相之间的依赖也会更困难，但凡有方法我们就要进行管理。

整个过程由于异步的脚本显得更加混乱。如果一个<script>标签标记有一个 async 属性，脚本的内容会在 HTTP 后台加载，加载完毕之后执行。脚本加载时，页面的其余部分，包括后续脚本的标签，将继续加载。当应用程序脚本进行解析和执行时，大量的异步加载的依赖（或者资源很慢的依赖）可能不可用。即使应用的<script>标签具有 async 属性，开发人员无法控制异步加载的资源按顺序加载，也无法保证期望的依赖层次模式。

提示 HTML 5 的<script>标签的 defer 属性类似于 async，只不过是直到页面解析完毕才被执行。这些属性都会减少页面渲染延迟，从而提高用户体验和页面性能。这对于移动设备尤其重要。

RequireJS 生来就是为了解决这种业务依赖的流程问题，它提供给开发者一个标准的方式书写 JavaScript 模块（“脚本”），这些模块会在被执行前声明其本身的模块依赖关系。通过声明所有的依赖关系，RequireJS 可以确保整体依赖层次异步加载，并且按正确的顺序执行模块。这种模式被称为异步模块加载机制（AMD），与 Node.js 和 Browserify 的模块载入机制所依赖的 CommonJS 模块载入机制形成了鲜明的对比。这两种方式在不同的使用场景中都会发挥不同的作用，RequireJS 和 AMD 主要用于解决 Web 浏览器端的问题以及 DOM 的缺点问题。实际上，使用 RequireJS 还是使用 Browserify，这种优先级往往是由工作流或者团队已使用的工具的开发模式决定的。

例如，RequireJS 可以为一些必须加载的但是非 AMD 模式的依赖创建兼容的垫片代码（这些库通常是远程分发网络上的或者旧代码）。这很重要，因为 RequireJS 假设 Web 应用程序中的脚本可能来自多个源，并且不会直接被开发者控制。默认情况下，RequireJS 不将所有应用程序脚本“打包”成一个单一文件，而是发出 HTTP 请求加载每个脚本。RequireJS 的工具 r.js（后续将会提到）为生产环境提供了打包

的聚合文件，但是仍然可以从别的位置加载远程的、已兼容的脚本。从另一方面来讲，Browserify 提供了“优先打包”的方法。它假定所有的内部脚本和依赖关系将被打包成一个单独的文件，其他远程的脚本会被分别加载。这会使远程脚本超出 Browserify 的控制，但是像 CommonJS 模式的 **bromote** 的插件是在打包的过程中加载远程文件。对于这两种方法，最终的结果是相同的：远程资源可以在运行时提供给应用程序。

5.1 运行示例

本章包含了可以在现代浏览器中运行的各种实例。在安装依赖或者运行 Web 服务器脚本时，Node.js 必不可少。为了安装示例代码的依赖，在终端打开 `code/requirejs` 的目录，运行命令 `npm install`。这个命令会读取 `package.json` 文件的内容并下载运行每个示例所必不可少的包。

本章的示例代码块都在文件的顶部包含了一段注释。例如，清单 5-1 的虚拟 `index.html` 文件会在 `example-000/public/` 路径下被找到。（这个路径并不是真实存在的，所以你如果没找到，请不必担心。）

清单 5-1 一个令人兴奋的 HTML 文件

```
<!-- example-000/public/index.html -->
<html>
  <head></head>
  <body><h1>Hello world!</h1></body>
</html>
```

除非另有说明，我们假设所有的示例代码目录均包含一个 `index.js`，用于启动最基本的 Web 服务器。清单 5-2 演示了在一个终端中如何用 Node.js 运行虚拟的 Web 服务器脚本 `example-000/index.js`。

清单 5-2 启动一个令人兴奋的 Web 服务器

```
example-000$ node index.js
>> mach web server started on node 0.12.0
>> Listening on :::8080, use CTRL+C to stop
```

命令的输出信息说明 Web 服务器正在监听 `http://localhost:8080`。在 Web 浏览器中打开 `http://localhost:8080/index.html` 网页，清单 5-1 中的 HTML 片段就会渲染出来。

5.2 使用 RequireJS

在 Web 应用程序中使用 RequireJS 工作流程通常包含一些常用的步骤。首先，RequireJS 必须在 HTML 文件内使用 `<script>` 标签加载。RequireJS 可引用自 Web 服务器或 CDN 上的独立脚本，也可以通过包管理器如 Bower 和 npm 来安装，然后通过本地 Web 服务器加载。其次，RequireJS 必须配置，让它知道脚本和模块的位置、如何为不符合 AMD 模式的脚本做兼容、加载哪些插件等。一旦配置结束，RequireJS 会加载主要应用模块，负责加载主要的页面组件，基本上“启动”页面的应用程序代码。此时 RequireJS 计算模块创建的依赖树，并开始在后台异步加载依赖脚本。一旦所有的模块载入完毕，应用代码会继续处理所有其处理范围内的事情。

这个过程的每一步都在下面的章节给出了详细的解释。每一部分的示例代码都展示了应用程序

从简单到复杂的过程，这种过程也能体现出一些（半）著名的人对此鼓舞人心并且幽默的思考。

5.2.1 安装

RequireJS 脚本文件可以直接从 <http://requirejs.org/> 网站下载。它分为几个不同的版本：一种是普通的 RequireJS 脚本，一种是 RequireJS 与 jQuery 和 Node.js 包预先绑定在一起的，该 Node.js 包包含了 RequireJS 本身和打包工具 r.js。本章中大部分示例用的都是普通版的 RequireJS。预捆绑 jQuery 脚本仅仅是为开发人员提供了方便。如果你想要在一个已经有 jQuery 库的项目中添加 RequireJS，普通版的 RequireJS 可以顺畅地兼容已安装的 jQuery——尽管旧版本的 jQuery 可能需要 shim 一下（shim 的脚本将在后面介绍）。

一旦获得了该脚本，RequireJS 的脚本会在 Web 应用里被一个 `<script>` 标签引用。RequireJS 是一个模块加载器，它承担加载所有其他 JavaScript 文件和应用程序可能需要的模块的责任。因此，极有可能 RequireJS 的 `<script>` 标签将是页面中唯一的一个 `<script>` 标签。清单 5-3 中给出了一个简单的示例。

清单 5-3 在 Web 页面中包含 RequireJS 脚本

```
<!-- example-001/public/index.html -->
<body>
  <header>
    <h1>Ponderings</h1>
  </header>
  <script src="/scripts/require.js"></script>
</body>
```

5.2.2 配置

RequireJS 脚本在页面中加载完毕后，就会开始寻找一个配置，这个配置主要告诉 RequireJS 脚本和模块的位置。配置选项可以由下面三种方式的任意一种提供。

首先，在 RequireJS 脚本加载完毕之前会创建全局的 `require` 对象。这个对象会包含所有 RequireJS 配置选项。此外，RequireJS 一旦加载完所有应用的模块，会执行一个“开始”的回调。

清单 5-4 中的脚本代码块显示了一个新创建的 RequireJS 配置对象，其存储在全局的 `require` 变量中。

清单 5-4 使用全局 `require` 对象配置 RequireJS

```
<!-- example-001/public/config01.html -->
<body>
  <header>
    <h1>Ponderings</h1>
  </header>
  <section id="quotes"></section>
  <script>
    /*
     * 会以 window.require 的形式自动附加在全局 window 对象上。
     */
    var require = {
      // 配置
      baseUrl: '/scripts',
      // 启动
      deps: ['quotes-view'],
      callback: function(quotesView) {
        quotesView.addQuote('Lorem ipsum dolor sit amet, consectetur adipiscing elit.');
```

```

        quotesView.addQuote('Nunc non purus faucibus justo tristique porta.');
```

这个对象中最重要的配置属性 `baseUrl`，确定一个相对于应用程序根目录的路径，然后 `RequireJS` 就可以开始解决模块依赖。`deps` 数组指定模块，会在配置之后被立即加载。`callback` 方法的存在是为了依赖的模块被加载后接收这些模块。这个示例加载了一个单一模块 `quotes-view`。一旦回调函数被调用，它可以访问这个模块的属性和方法。

清单 5-5 的路径树展示了 `quotes-view.js` 文件相对于 `config-1.html`（即将被访问的页面）和 `RequireJS` 文件的位置。

清单 5-5 应用文件位置

```

├── config01.html
├── scripts
│   ├── quotes-view.js
│   └── require.js
├── styles
└── app.css
```

注意，`quotes-view` 模块的绝对路径和文件扩展名在 `deps` 数组中是被忽略的。默认情况下，`RequireJS` 会假设任何给定的模块位于正在查看的页面的相对路径下，每个模块都被包含在单个的 `JavaScript` 文件中。在这种情况下，后者假设是真的，但前者不是，这就是为什么制定一个 `baseUrl` 是必要的。当 `RequireJS` 试图解析任何模块时，它会把任意 `baseUrl` 值和模块名合并在一起，然后追加 `.js` 文件扩展名来产生一个相对于应用程序的根路径的完整路径。

当 `config01.html` 页面加载时，通过 `quotesView.addQuote()` 方法传递的字符串会显示在页面上。

第二种配置方法类似于第一种，但是在 `RequireJS` 脚本加载完后，使用 `RequireJS` 的 API 来写配置，如清单 5-6 所示。

清单 5-6 利用 `RequireJS` 的 API 来配置

```

<!-- example-001/public/config02.html -->
<body>
  <header>
    <h1>Ponderings</h1>
  </header>
  <section id="quotes"></section>
  <script src="/scripts/require.js"></script>
  <script>
    // configuration
    requirejs.config({
      baseUrl: '/scripts'
    });
    // kickoff
    requirejs(['quotes-view'], function (quotesView) {
      quotesView.addQuote('Lorem ipsum dolor sit amet, consectetur adipiscing elit.');
```

在本例中，`<script>` 模块首先使用由 `require.js` 脚本创建的全局对象，调用其 `config()` 方法可配置 `RequireJS`。然后它会调用 `requirejs` 来启动这个应用。传递给 `config()` 方法的对象类似清单 5-4

中的全局 `require` 对象，但没有 `deps` 和 `callback` 属性。取而代之，`requirejs` 的方法通过接受一个应用的依赖数组和一个回调方法。当模块设计在后面介绍了以后，这种方式会变得非常熟悉。

最终结果的作用是相同的：`RequireJS` 用它的配置去加载 `quotes-view` 模块。一旦这个模块加载完毕，回调函数会与它一起与页面产生交互。

第三种配置方法使用第二种方式的语法，但是把配置和启动代码移动到它本身的脚本中。`RequireJS` 的 `script` 标签中用 `data-main` 属性来告诉 `RequireJS` 其配置和开始模块在哪里（见清单 5-7）。

清单 5-7 通过一个额外的脚本来配置 `RequireJS`

```
<!-- example-001/public/config03.html -->
<body>
  <header>
    <h1>Ponderings</h1>
  </header>
  <section id="quotes"></section>
  <script src="/scripts/require.js" data-main="/scripts/main.js"></script>
</body>
```

一旦 `RequireJS` 加载完毕，它将寻找 `data-main` 属性。如果找到该属性，就会异步地加载属性值中指定的脚本。清单 5-8 展示了 `main.js` 的内容，它与清单 5-6 里的 `<script>` 代码块是等价的。

清单 5-8 `RequireJS` 的主要模块

```
// example-001/public/scripts/main.js
// 配置
requirejs.config({
  baseUrl: '/scripts'
});

// 启动
requirejs(['quotes-view'], function (quotesView) {
  quotesView.addQuote('Lorem ipsum dolor sit amet, consectetur adipiscing elit.');
```

注意 由于 `data-main` 脚本是异步加载，`RequireJS` 后紧接着的脚本或者包含的 `script` 代码块可能会先执行。如果 `RequireJS` 管理应用中的所有脚本，或者如果加载后 `RequireJS` 脚本与应用程序本身（如广告脚本）没有关系，将不会有任何冲突。

5.2.3 应用模块和依赖

`RequireJS` 的模块是通过以下三个要素定义的：

1. 一个模块名；
2. 一个依赖列表（众多模块）；
3. 一个模块，其闭包内能将每个依赖模块的输出作为函数参数，创建模块代码，并可能返回其他模块可以使用的东西。

清单 5-9 展示了一个虚拟模块中的这几个要素。全局的 `define()` 方法调用后，模块就被创建了。这个方法接受三个参数，对应上述三点。

清单 5-9 模块剖析

```
define( /*#1*/ 'm1', /*#2*/ ['d1', 'd2'], /*#3*/ function(d1, d2) {
```



```

/*
 * 定义在模块闭包内的变量是模块的
 * 私有变量，并且不会暴露给其他模块。
 */
var privateModuleVariable = "can't touch this";

/*
 * 若某模块指定了 m1 为依赖，那么
 * 返回值(如果有返回值)可被该模块使用。
 */
return {
  getPrivateModuleVariable: function() {
    return privateModuleVariable;
  }
};
})

```

模块本身的名字很关键。在清单 5-9 中，m1 是被显式声明的。如果未指定模块名称（只通过依赖关系和模块闭包作为唯一的参数进行定义），那么 RequireJS 将假定模块的名称就是包含该模块的脚本文件名（除去.js 扩展名）。这种做法在实际中很常见，但为了明确阐释，本处展示了模块名。

提示 给模块具体的名称有可能引入不必要的复杂性，因为 RequireJS 的加载模块取决于脚本 URL 路径。如果显示定义了模块名称，并且文件名不匹配模块名，那么 RequireJS 的配置文件中需要定义模块别名到模块实际 JavaScript 文件名的映射。这将在下一节中进行讨论。

清单 5-9 的依赖列表指定了 RequireJS 需要加载的其他两个模块。d1 和 d2 是这两个模块的模块名，指定在脚本文件 d1.js 和 d2.js 中。这些脚本和清单 5-9 定义的模块很相似，但它们会加载各自的依赖。

最后，模块闭包接受每个依赖模块的输出作为函数参数。此输出可以是每个依赖模块的闭包中返回的任何值。清单 5-9 中的闭包返回了它自己的值。如果另外一个模块把 m1 声明为其依赖，那么这个返回值将传递到那个模块的闭包中。

如果一个模块没有依赖关系，它的依赖组将是空的，并且模块闭包不会收到任何参数返回值。

当一个模块被加载后，将一直保存在内存中，直到应用程序终止。如果多个模块声明了相同的依赖，那么该依赖只需被加载一次。无论其闭包返回什么值，都将传递到两个模块中。其中任何一个给定模块将与其他所有的模块共享。

一个模块可以返回任何有效的 JavaScript 值，或者如果该模块仅仅是为了操作其他模块而存在的、抑或是仅为了产生副作用，则完全什么都不返回。

清单 5-10 显示了 example-002/public 目录的结构。该结构类似于 example-001，但是添加少许附加模块，名为 data/quotes.js（抓取报价数据模块）和 util/dom.js（一个封装了全局 window 对象的模块，以便它们不需要直接访问 window）。

清单 5-10 example-002 的公共路径结构

```

public
├── index.html
├── scripts
│   ├── data
│   │   ├── quotes.js
│   │   ├── main.js
│   │   ├── quotes-view.js
│   │   ├── require.js
│   │   ├── util
│   │   └── dom.js

```

回想一下，一个模块的依赖是相对于 RequireJS 的 baseUrl 值存在的。当一个模块指定了依赖路径，它是相对于 baseUrl 的路径。在清单 5-11 中，main.js 文件依赖了 data/quotes 模块（public/scripts/data/quotes.js），而 quotes-view.js 模块依赖了 util/dom（public/scripts/util/dom.js）。

清单 5-11 模块依赖路径

```
// example-002/public/scripts/main.js
requirejs(['data/quotes', 'quotes-view'], function(quoteData, quotesView) {
    // ...
});

// example-002/public/scripts/data/quotes.js
define([/*no dependencies*/ ], function() {
    // ...
});

// example-002/public/scripts/quotes-view.js
define(['util/dom'], function(dom) {
    // ...
});

// example-002/public/scripts/util/dom.js
define([/*no dependencies*/ ], function() {
    // ...
});
```

如图 5-1 所示，当这些模块被加载的时候，依赖树被创建了。

随着应用的依赖增多，模块路径会变得冗长，令人乏味。然而，有两种方式缓解这个问题。首先，一个模块可以使用前导点符号指定相对于自身的依赖。例如，模块声明了一个依赖./foo 会加载同级的 foo.js 文件，定位在与其自身相同的 URL 片段上，而如果模块的依赖是../bar，则会向与自身相同的 URL 段的上一级去寻找 bar.js，这大大减少了依赖冗长。

其次，模块可以用路径别名来命名，定义在 RequireJS 配置中，如在下一节中所述。

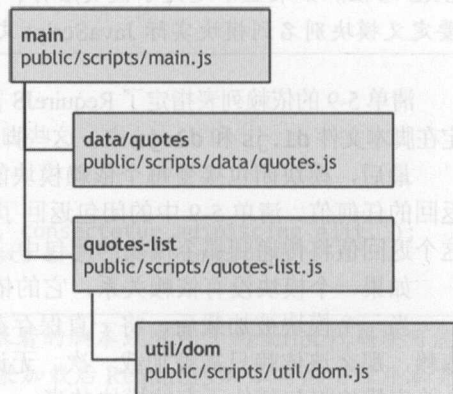


图 5-1 RequireJS 依赖树

5.2.4 路径和别名

为模块分配别名，使得其他模块可以使用别名作为一个依赖的名字，而不是完整的模块路径名。基于种种原因来说，这都是有用的，但通常用来简化模块路径，消除路径中的版本号，或者版本名称，从而显示地声明自己的模块的名称。

清单 5-12 的模块中依赖了一个三方库 jQuery。如果 jQuery 模块位于/scripts/jquery.js，不需要模块别名就可以加载这个依赖；RequireJS 会基于 baseUrl 配置的值来定位文件。

清单 5-12 声明一个 jQuery 模块依赖

```
define(['jquery'], function ($) {
    // ...
});
```

然而, `jquery` 不见得就被定义在 `baseUrl` 配置的根路径。很有可能 `jquery` 脚本会存在于一个 `vendor` 的路径下, 比如 `/scripts/vendor/jquery`, 而且 `jQuery` 的库的名称会包含 `jQuery` 的版本号 (如 `jquery-2.1.3.min`), 正如 `jQuery` 脚本的发布方式。更糟糕的是, `jQuery` 显示声明自己的模块的名称 `jquery`。如果一个模块尝试使用完整路径 `/scripts/vendor/jquery/jquery-2.1.3.min` 加载 `jquery` 脚本, `RequireJS` 会通过 `HTTP` 加载脚本, 结果是导入模块失败, 因为它声明的名字是 `jquery`, 不是 `jquery-2.1.3.min`。

■ **提示** 显示命名模块被认为是不良习惯, 因为应用程序模块必须使用模块声明的名字, 包含该模块的脚本文件必须在 `RequireJS` 的配置里分配其名称或者别名。为 `jQuery` 有一个特殊的让步, 是因为 `jQuery` 是一个无处不在的库。

别名在 `RequireJS` 配置散列中的 `paths` 属性中指定。别名是 `RequireJS` 配置中在 `paths` 属性下散列声明的。在清单 5-13 中, 别名 `jquery` 被指向 `vendor/jquery/jquery-2.1.3.min`, 这个路径是相对于 `baseUrl` 的。

清单 5-13 配置模块路径别名

```
requirejs.config({
  baseUrl: '/scripts',
  // ... 其他配置 ...
  paths: {
    'jquery': 'vendor/jquery/jquery-2.1.3.min'
  }
});
```

在 `paths` 的对象中, 别名是键, 映射的脚本路径是值。一旦一个模块的别名被定义, 它也可以在其他模块依赖列表中被使用。清单 5-14 展示了 `jquery` 别名的使用。

清单 5-14 在依赖清单中使用一个模块的别名

```
// jquery 这个别名指的是 vendor/jquery/jquery-2.1.3.min
define(['jquery'], function ($) {
  // ...
});
```

因为模块别名优先于实际模块位置, `RequireJS` 会在尝试定位 `/scripts/jquery.js` 之前, 就解析 `jQuery` 脚本的路径。

■ **注意** 匿名模块 (不声明自己的模块名称), 可以把任何模块名当作别名。但如果想为命名模块定义别名 (如 `jquery`), 那么必须使用自身已经声明的模块名作别名。

使用代理模块加载插件

一些库像 `jQuery`、`Underscore`、`Lodash`、`Handlebars` 等都有插件系统来让开发人员扩展功能。模块别名策略可以实际地帮助开发者一次性为这些库加载扩展, 而不需要指定每个模块的扩展来使用它们。在清单 5-15 中, `jquery` 脚本位置是名为 `jquery` 的别名来定义。为简洁起见, 自定义模块 `util/jquery-all` 使用 `jquery-all` 作为别名。所有的应用模块会通过声明 `jquery-all` 作为依赖而加载 `jquery`。通过 `jquery-all` 模块加载正常的 `jquery` 模块, 然后为其高度自定义插件。

清单 5-15 使用模块别名来加载 jQuery 插件

```
requirejs.config({
  baseUrl: '/scripts',
  // ... 其他配置 ...
  paths: {
    // 基础脚本
    'jquery': 'vendor/jquery/jquery-2.1.3.min',
    // 自定义扩展
    'jquery-all': 'util/jquery-all'
  }
});

// example-003/public/scripts/util/jquery-all
define(['jquery'], function($) {

  $.fn.addQuotes = function() { /*...*/ };

  return $;
  // 或者
  //return $.noConflict(true);
});
```

jquery-all 这个代理模块返回了 jQuery 对象本身,它允许模块依赖 jquery-all 访问 jQuery 加载自定义扩展。默认情况下, jQuery 会注册在全局对象 window 上——即便它可能被当作 AMD 模块来使用。如果所有应用的模块都通过 jquery-all 模块来访问 jQuery (甚至是纯净的 jquery 模块,就像大多数基础库的做法),然后就不须把 jQuery 作为全局变量。全局 jquery 对象可以通过调用 \$.noConflict(true) 被移除。这将返回 jquery 对象,并且在清单 5-15 中作为 jquery-all 模块的替代返回值。

由于 jQuery 现在是示例应用的一部分,负责在 DOM 上呈现名人名言数据的 quotes-view 模块,不需要再依赖 util/dom 模块。它可以指定 jquery-all 作为依赖,一次性地加载 jquery 和自定义 addQuotes() 插件方法。清单 5-16 展示了 quotes-view 模块的变化。

清单 5-16 在 quotes-view 模块中加载 jQuery 和自定义插件

```
// example-003/public/scripts/quotes-view.js
define(['jquery-all'], function($) {
  var $quotes = $('#quotes');

  return {
    render: function(groupedQuotes) {
      for (var attribution in groupedQuotes) {
        if (!groupedQuotes.hasOwnProperty(attribution)) continue;
        $quotes.addQuotes(attribution, groupedQuotes[attribution]);
      }
    }
  };
});
```

使用一个模块代理下载 jquery 的优点是,对同时依赖 jquery 和自定义插件的其他模块来说,不再需要在依赖中同时指定这两个模块。如果没有这种技术,应用模块都需要多个依赖来保证 jQuery 会在合适的时间加载合适的插件,如清单 5-17 所示。

清单 5-17 不使用代理模块来加载插件

```
// scripts/util/jquery-plugin-1.js
define(['jquery'], function($) {
```



```

$.fn.customPlugin1 = function() { /*...*/ };
});

// scripts/util/jquery-plugin-2.js
define(['jquery'], function($) {
    $.fn.customPlugin2 = function() { /*...*/ };
});

// scripts/*/module-that-uses-jquery.js
define(['jquery', 'util/jquery-plugins-1', 'util/jquery-plugins-2'], function($) {
    // ...
});

```

在这种情况下，即使 `jquery-plugin-1` 和 `jquery-plugin-2` 不返回值，它们也必须被添加为依赖，以至于向 `jquery` 模块添加多个依赖的副作用还会发生。

5.2.5 Shims

支持 AMD 模式的模块是可以直接使用 RequireJS 的。不支持 AMD 模式的库也可以使用，但是需要配置 RequireJS 的 `shim` 属性，或者手动创建一个已兼容的模块。`example-003` 的 `data/quotes` 模块中暴露了一个 `groupByAttribution()` 方法来遍历名人名言的集合。它创建了一个哈希表，其中的 `key` 是人名，`value` 是对应的名言的数组。这种组函数便于其他地方引用。

幸运的是，一个叫 `undrln` 的库可以提供广义上说类似的这种功能，但它不是 AMD 兼容版。其他的 AMD 模块使用 `undrln` 作为依赖对于一个 `shim` 模块来说是必要的，`undrln` 写在一个闭包的标准 javascript 模块中，如清单 5-18 所示。它将自己分配给全局 `window` 对象，允许其他脚本在页面上对它进行访问。

■ **注意** 该 `undrln.js` 脚本公然模仿了不兼容 AMD 模式 `Lodash` API 的一个子集，专门用作这一章的例子。

清单 5-18 完全原创的 `Undrln` 库

```

// example-004/public/scripts/vendor/undrln/undrln.js
/**
 * undrln (c) 2015 133th@x0r
 * MIT license.
 * v0.0.0.0.1-alpha-DEV-theta-r2
 */
(function () {

    var undrln = window._ = {};

    undrln.groupBy = function (collection, key) {
        // ...
    };

})();

```

创建 `shim` 的时候有几点是必须被添加到 RequireJS 的配置项中的：第一，必须在 `paths` 的属性下创建一个别名，以便让 RequireJS 知道被 `shim` 的模块在哪儿；第二，`shim` 配置入口必须添加到相应 `shim` 模块。在清单 5-19 中，这两点都被添加到了 RequireJS 的配置中。

清单 5-19 模块的 shim 配置

```
// example-004/public/scripts/main.js
requirejs.config({
  baseUrl: '/scripts',
  paths: {
    jquery: 'vendor/jquery/jquery-2.1.3.min',
    'jquery-all': 'util/jquery-all',
    // 给 undrln 起一个模块别名
    undrln: 'vendor/undrln/undrln'
  },
  shim: {
    // 为 undrln 定义 shim
    undrln: {
      exports: '_'
    }
  }
});
```

在 shim 字段下的每个 key 定义的是需要被添加垫片的模块别名，分配给这些 key 的对象指定了该垫片如何工作。使用这个钩子，RequireJS 通过定义一个空的 AMD 模块，由普通脚本或者一个三方库返回了一个 global 对象来创建一个 shim。undrln 创建了全局的 window._ 对象，因此在 shim 的配置中指定 undrln 导出的模块名叫 _。最后，生成的 RequireJS 的 shim 模块看起来像是清单 5-20 所示的模块。请注意，这些垫片在模块加载时被动态创建，也就是说，这些模块并不是真实存在于 Web 服务器上的文件。（有一个例外是 r.js 打包工具，生成的 shim 模块输出到一个包文件中，作为一种优化措施，这将在稍后讨论。）

清单 5-20 RequireJS 的 shim 模块示例

```
define('undrln', [], function () {
  return window._;
});
```

在清单 5-21 中，名人名言模块示例使用加了垫片的 undrln 作为依赖。

清单 5-21 RequireJS 的 shim 模块示例

```
// example-004/public/scripts/data/quotes.js
define(['undrln'], function (_) {
  //...
  return {
    groupByAttribution: function () {
      return _.groupBy(quoteData, 'attribution');
    },
    //...
  }
});
```

通过对不支持 AMD 的脚本进行 shim，当 AMD 模块依赖了非 AMD 模式的脚本时，RequireJS 可利用其异步加载的功能在后台加载脚本。倘若没有这种功能，这些脚本就需要在每个页面用 <script> 标签异步加载，以确保可用性。运行示例 example-004 这个 Web 应用，然后在浏览器中访问 <http://localhost:8080/index.html> 页面，这个页面将显示一个名人名言列表。图 5-2 展示了渲染的页面，Chrome 中列出了所有已加载 JavaScript 模块的 Network 面板。可以看到，Initiator 这一列清楚展示了 RequireJS 在负责加载所有模块，甚至 undrln.js 这个不支持 AMD 模式的模块也被展示在列表中。

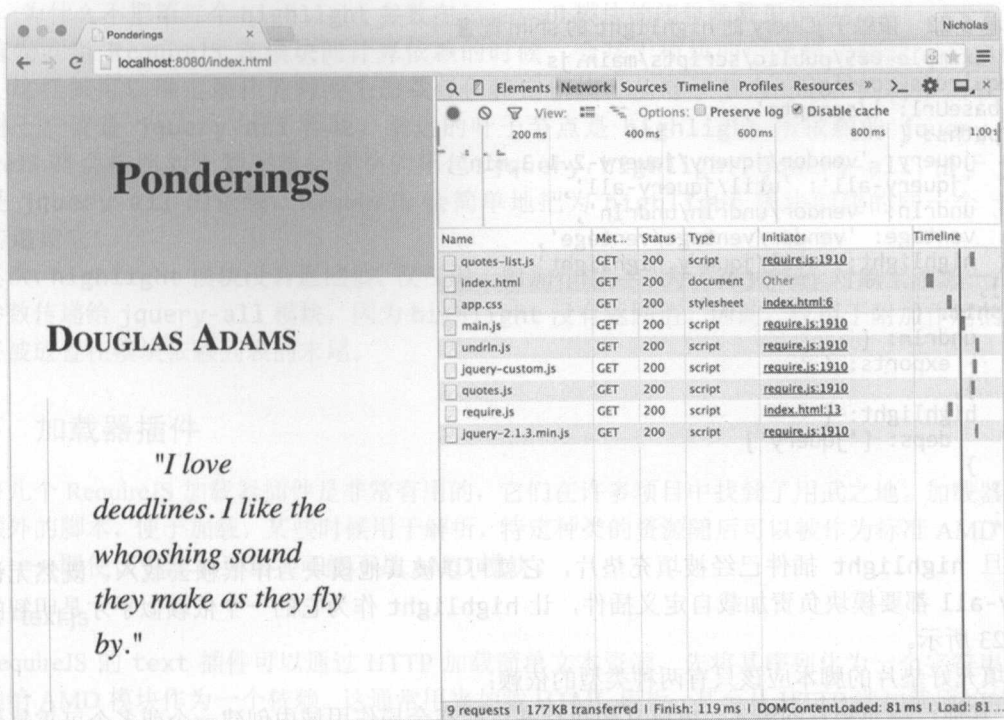


图 5-2 Chrome 中显示的 RequireJS 加载的模块

shim 依赖

为已填充垫片的模块声明依赖是非常合理的一件事，如全局作用域包含的对象。当 AMD 模块声明了依赖，RequireJS 确保了依赖是在代码执行前被先载入的。已填充垫片模块的依赖是在 shim 的配置文件以类似方式声明的。已填充垫片模块有可能依赖其他已填充垫片模块，甚至是 AMD 模式的模块，前提是这些 AMD 模块能在全局作用域中被允许访问（通常是一个坏办法，但是有时候非用不可）。

为了丰富示例应用，在 example-005 的名人名言页面中增加一个搜索框。在搜索栏中输入的文本只要在任何名人引言中，都会高亮显示。目前为止，所有的示例用的都是一个视图 quotes-view，来展示被渲染的标记。由于应用程序的功能还在扩展，需要引入两个模块来管理功能：search-view 和 quotes-state。search-view 模块用来负责监视用户输入的文本字段。当搜索内容改变后，这个视图通知 quotes-state 模块搜索动作已发生，并把改变的值传给它。quotes-state 作为所有视图的单一状态源，当它接收一个新的搜索项的时候，会触发一个事件来通知已被订阅的视图。挖掘一些老旧的源码文件 public/scripts/util/jquery.highlight.js，它是一个可以在 DOM 模式中高亮文本的非 AMD 模式 jQuery 插件。当 quotes-view 模块从 quotes-state 接收搜索事件，将会调用这个插件把 quotes-state 传过来的文本内容高亮显示。为了使用这段旧的脚本，path 和 shim 都添加到 main.js 配置中。highlight 插件没有导出任何值，但是需要 jQuery 在它之前加载，否则就会在尝试访问全局 jQuery 对象时抛出错误。如清单 5-22 所示，依赖已经通过 deps 属性添加到 highlight 的 shim 对象中。这个属性包含一个数组，数组内容是模块的名字或者别名，这些模块会在加载当前 shim 模块之前载入——就如同本例中的 jQuery 一样。

清单 5-22 依赖于 jQuery 的 highlight 的 shim 配置

```
// example-005/public/scripts/main.js
requirejs.config({
  baseUrl: '/scripts',
  paths: {
    jquery: 'vendor/jquery/jquery-2.1.3.min',
    'jquery-all': 'util/jquery-all',
    undrln: 'vendor/undrln/undrln',
    ventage: 'vendor/ventage/ventage',
    highlight: 'util/jquery.highlight'
  },
  shim: {
    undrln: {
      exports: '_'
    },
    highlight: {
      deps: ['jquery']
    }
  }
});
```

一旦 **highlight** 插件已经被填充垫片，它就可以被其他模块当作依赖去载入。既然无论如何 **jquery-all** 都要模块负责加载自定义插件，让 **highlight** 作为它的一个依赖似乎才是明智的，如清单 5-23 所示。

被填充好垫片的脚本应该只有两种类型的依赖：

- 其他已填充垫片的脚本，可以立即执行并可能在全局作用域内创建一个或多个可重复使用的变量或命名空间
- AMD 模块，同样在全局作用域中创建了可重复使用的变量或命名空间（伴随着副作用）。

由于 AMD 模块通常并不跟全局作用域打交道，把这种模块作为已填充垫片脚本的依赖几乎是没用的，因为没有办法让一个被 shim 的脚本访问一个 AMD 模块的 API。如果一个 AMD 模块没有向全局作用域添加内容，那么对于已填充垫片脚本来说是没用的。同样，AMD 的模块是异步加载并且在特定序列的闭包中被执行的（下一节讨论），然而已填充垫片脚本一旦加载就会执行。（记住：已填充垫片脚本只是普通的脚本，一旦被引入 DOM 中就会执行。非 AMD 脚本中生成的已填充垫片模块仅简单地传递全局输出给依赖它的 AMD 模块。）即使某个已填充垫片脚本可以访问某个 AMD 模块的 API，也不能保证在该脚本执行的时候可以访问到。

清单 5-23 作为其他模块的依赖来加载 highlight 模块

```
// example-005/public/scripts/util/jquery-all.js
define(['jquery', 'highlight'], function ($) {

  $.fn.addQuotes = function (attribution, quotes) {
    // ...
  };

  return $;
});
```

采用这种方案可能会立刻想到两个问题：

1. **highlight** 和 **jquery-all** 两个模块都声明了 **jquery** 作为依赖，那么 jQuery 实际是什么时候加载的呢？

2. 为什么不把第二个 `highlight` 参数在 `jquery-all` 模块的闭包函数里声明？

首先，当 `RequireJS` 在模块间计算依赖的时候，会基于模块的层次模式创建一个内部的依赖树。这样做可以确定最佳的时间来加载任何特定模块，从叶子节点开始，向树干加载。本例中“树干”就是 `jquery-all` 模块，最远的叶子节点是 `highlight` 所依赖的 `jquery` 模块。`RequireJS` 将会按照如下顺序执行模块的闭包：`jquery`, `highlight`, `jquery-all`。由于 `jquery` 同样是 `jquery-all` 的依赖，`RequireJS` 会简单地把为 `highlight` 模块创建的同一个 `jquery` 实例传递给它。

其次，`highlight` 模块没有返回值，仅仅用于附加作用——为了在 `jQuery` 对象上添加一个插件。没有参数传递给 `jquery-all` 模块，因为 `highlight` 没有返回值。因此，仅用于附加作用的依赖应该始终被放置在模块依赖列表的末尾。

5.2.6 加载器插件

有几个 `RequireJS` 加载器插件是非常有用的，它们在许多项目中找到了用武之地。加载器插件是一个额外的脚本，便于加载，某些时候用于解析，特定种类的资源随后可以被作为标准 AMD 模块依赖引入——即使该资源本身其实可能不是 AMD 模块。

1. `text.js`

`RequireJS` 的 `text` 插件可以通过 HTTP 加载简单文本资源，先将其序列化为一个字符串，并将其传递给 AMD 模块作为一个依赖。这通常用来加载 HTML 模板，甚至从 HTTP 端加载原始的 JSON 数据。要安装这个插件，`text.js` 必须从项目的仓库中复制，并按照惯例，与 `main.js` 的配置文件放在同一个路径下。（其他安装方法都列在插件项目的 README 文件中。）

示例应用的 `quotes-view` 模块使用一个 `jQuery` 的插件来构建人名名言列表，一次一个 DOM 元素。这种方式效率不高，而且很容易用模板方案取代。兼容 AMD 的模板库 `Handlebars` 可以很好地用于这项工作。模板构建的库，针对这种任务是非常好的选择。清单 5-24 中，添加这个库到 `example-006` 的 `vendor` 路径下，并且 `main.js` 的配置文件中有了简洁的模块别名。

清单 5-24 `Handlebars` 模块别名

```
// example-006/public/scripts/main.js
requirejs.config({
  baseUrl: '/scripts',
  paths: {
    //...
    Handlebars: 'vendor/handlebars/handlebars-v3.0.3'
  },
  //...
});
```

当 `quotes-view` 模块渲染自身时，用的是哈希对象中的分人名言数据。该数据的 `key` 便是属性（例如：每条名言对应的人名），`value` 是每个人名对应的名言数组集合。（一个给定的属性可以与一个或多个名言相关联。）清单 5-25 展示了绑定这个数据片段的模板，模板文件位于 `public/scripts/templates/quotes.hbs`。

清单 5-25 `quotes-view` 的 `Handlebars` 模板

```
<!-- example-006/public/scripts/templates/quotes.hbs -->
{{#each this as |quotes attribution|}}
```

```
<section class="multiquote">
  <h2 class="attribution">{{attribution}}</h2>
  {{#each quotes}}
    <blockquote class="quote">
      {{#explode text delim="\n"}}
        <p>{{this}}</p>
      {{/explode}}
    </blockquote>
  {{/each}}
</section>
{{/each}}
```

为理解模板遍历数据对象，将每个属性和其相关联的引言抽取出来。你不必非常熟悉 Handlebars 的语法。其实并没有必要为了弄清楚模板如何通过数据对象进行渲染而对 Handlebars 的语法完全熟悉，数据对象输出的是其每个属性以及相关报价。模板为每个 attribution 创建了一个<h2>标签，然后为每条引述创建了<blockquote>元素来包裹其文本。有一个较特别的辅助方法。#explode，通过(n)分隔符对名人名言折行，然后把每个名言片段用<p>标签包裹起来。

explode 这个辅助方法是比较值得关注的，因为它并不是 Handlebars 原生的。它是在 Handlebars 辅助函数中定义并注册的，位于文件 public/scripts/util/handlebars-all.js，如清单 5-26 所示。

清单 5-26 Handlebars 辅助方法#explode

```
// example-006/public/scripts/util/handlebars-all.js
define(['Handlebars'], function (Handlebars) {
  Handlebars.registerHelper('explode', function (context, options) {
    var delimiter = options.hash.delim || '';
    var parts = context.split(delimiter);
    var processed = '';
    while (parts.length) {
      processed += options.fn(parts.shift().trim());
    }
    return processed;
  });
  return Handlebars;
});
```

由于这个模块增加了辅助方法然后返回了 Handlebars 对象，quotes-view 模块会将其作为依赖导入，代替普通的 Handlebars 模块，就像使用 jquery-all 模块来代替 jquery 模块使用一样。相应的模块别名也已经添加到清单 5-27 的配置中。

清单 5-27 handlebars-all 模块别名

```
// example-006/public/scripts/main.js
requirejs.config({
  baseUrl: '/scripts',
  paths: {
    //...
    Handlebars: 'vendor/handlebars/handlebars-v3.0.3',
    'handlebars-all': 'util/handlebars-all'
  },
  //...
});
```

在清单 5-28 中，quotes-view 模块已经被修改成引入 handlebars-all 模块和 quotes.hbs 模板。文本模板的模块名是非常奇怪的：它必须以 text!前缀开头，后面跟着模板文件相对于定义在 main.js 中的 baseUrl 的相对路径。

清单 5-28 quotes.hbs 作为一个模块依赖被导入

```
// example-006/public/scripts/quotes-view.js
define([
  'jquery-all',
  'quotes-state',
  'handlebars-all',
  'text!templates/quote.hbs'
],
function ($, quotesState, Handlebars, quotesTemplate) {

  var bindTemplate = Handlebars.compile(quotesTemplate);

  var view = {
    // ...
    render: function () {
      view.$el.empty();
      var groupedQuotes = quotesState.quotes;
      view.$el.html(bindTemplate(groupedQuotes));
    },
    // ...
  };
  // ...
});
```

当 RequireJS 遇到一个依赖的带有 **text!** 前缀的依赖名称时, 它会自动尝试载入 **text.js** 插件脚本, 然后作为一个字符串载入并序列化如上所述的文件。quotes-view 的回调函数中的 quotesTemplate 方法会包含 quotes.hbs 文件的序列化内容, 这些内容会被 Handlebars 编译并且用来在 DOM 中渲染模块。

2. 页面加载

当一个网页完全载入后, 会触发一个 **DOMContentLoaded** 事件 (在现代浏览器中)。在浏览器构建完 DOM 树之前载入的脚本经常会监听这个事件, 用于了解什么时候操作页面元素是安全的。如果脚本在结束的 **</body>** 之前载入, 它会假定大部分 DOM 已经被载入, 然后就不需要监听这个事件了。然而, 在 **<body>** 中的脚本或者常见的在 **<head>** 标签中的脚本就没有这么灵活的方式了。

即使 RequireJS 在结束标签 **</body>** 前被载入, 清单 5-29 中的 main.js 文件 (省略了一些配置) 仍然会在 **DOMContentLoaded** 被触发之后立即执行, 并传递一个方法给 jQuery。如果 RequireJS 的 **<script>** 标签被移动到文档的 **<head>** 标签中, 就不会发生这种情况了。

清单 5-29 利用 jQuery 确定 DOM 是否已经完全被载入

```
// example-006/public/scripts/main.js
// ...

requirejs(['jquery-all', 'quotes-view', 'search-view'],
function ($, quotesView) {
  $(function () {
    quotesView.ready();
  });
});
```

domReady 插件是一个奇特的加载器, 它在 DOM 完全准备好之后调用一个模块的回调。类似 text 插件, domReady.js 文件必须被定义在 main.js 中的 baseUrl 中, 要被 RequireJS 可访问到。

清单 5-30 展示了 `main.js` 修改后的版本（省略了一些配置）：`jquery` 的依赖已经被移除，`domReady!` 插件被添加到依赖列表中。尾随的感叹号告诉 RequireJS 这个模块作为一个插件加载器，而不是一个标准模块。不像文本插件，`domReady` 实际上什么都没载入，所以感叹号的后面实际上并不需要额外的信息。

清单 5-30 利用 `domReady` 插件确定 DOM 是否已经完全被载入

```
// example-007/public/scripts/main.js
// ...
```

```
requirejs(['quotes-view', 'search-view', 'domReady!'],
  function (quotesView) {
    quotesView.ready();
  });
```

3. i18n

RequireJS 通过 `i18n` 加载器插件支持国际化。（`i18n` 是一个基于数字的单词，数字 18 相当于单词 `internationalization` 里 `i` 和 `n` 之间的 18 个字母。）国际化是在编写一个 Web 应用的时候，以便适应其他国家和地区的用户使用（也被称为国家语言支持，或 NLS）。`i18n` 通常被用于翻译网站内控件的文本，或者 `chrome` 浏览器，如按钮、页头、超链接文本、文本区域等。为了展示这个插件的功能，示例应用需要添加两个新的模板：一个是页面的标题，一个是搜索框的占位符文本。实际的引文数据不会被翻译，因为可以假设该数据是从一个负责呈现合适译文的服务器中获取的。在本示例中，为简单起见，引文数据被硬编码在 `data/quotes` 模块中，并一直用英文显示。

在清单 5-31 中，`search.hbs` 模板也已经从文件 `index.html` 中提取出来，其为搜索栏接受的唯一输入是占位符文字，`search-view` 也已经做了适配，在渲染 DOM 中的内容的时候使用此模板。

清单 5-31 `search.hbs` 模板会显示搜索框占位符的译文

```
<!-- example-008/public/scripts/templates/search.hbs -->
<form>
  <fieldset>
    <input type="text" name="search" placeholder="{{searchPlaceholder}}"/>
  </fieldset>
</form>
```

清单 5-32 所示的是新增的 `header.hbs` 模板。它能够被新的 `header-view` 模板显示，这个模板接受的唯一输入就是页面标题。

清单 5-32 `header.hbs` 模板会显示页面标题的译文

```
<!-- example-008/public/scripts/templates/header.hbs -->
<h1>{{pageTitle}}</h1>
```

清单 5-33 中的 `header-view` 模块不仅展示了模板依赖是如何通过 `text` 插件导入的，也展示了语言模板如何通过 `i18n` 插件导入。与常见的加载器的语法几乎是一样的，插件名称后跟着一个感叹号和一个跟 `baseUrl` 相对的模块路径，本例中相对路径是 `nls/lang`。当模板被加载之后，被序列化之后的字符串会被传递到模板回调方法中，但是 `i18n` 插件加载的语言模块包含的是被翻译的文本数据，并把这个数据传递到模板的回调方法中。在清单 5-33 中，这个对象可以通过 `lang` 参数访问到。

清单 5-33 依赖于 i18n 语言对象的 header-view 模块

```
// example-008/public/scripts/header-view.js
define([
  'quotes-state',
  'jquery-all',
  'handlebars-all',
  'text!templates/header.hbs',
  'i18n!nls/lang'
], function (quotesState, $, Handlebars, headerTemplate, lang) {
  // ...
});
```

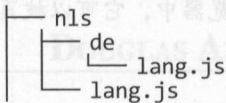
语言模块是一个普通的 AMD 模块。不同于像 `define()` 函数传递一个依赖列表和一个函数闭包，取而代之，它简单地使用对象字面量。这个对象字面量遵循一种非常明确的语法，如清单 5-34 所示。

清单 5-34 默认的英语语言模块

```
// example-008/public/scripts/nls/lang.js
define({
  root: {
    pageTitle: 'Ponderings',
    searchPlaceholder: 'search'
  },
  de: true
});
```

首先，当插件解析好语言译文后，`root` 字段存储的键值对用来获取译好的数据。这个对象的 `key` 是可以在编程的时候用来获取翻译后的文本内容。在 `search` 模板中，如 `{{searchPlaceholder}}` 区域的内容，在模板被绑定的时候，会被替换成语言对象中的 `searchPlaceholder` 字段的值。其次，`root` 属性的兄弟属性是一种 IETF（Internet 工程任务组）语言标记。这个值用来标识根据浏览器的默认语言设置，是否需要翻译。在本例中，德语的 `de` 语言标签被赋值为 `true`；如果是西班牙语，需要翻译，那么可以增加一个 `es-es` 属性，值被设置成 `true`。如果是法语的翻译，可以增加 `fr-fr` 属性，其他语言也是类似。当语言模块中新增一种语言需要翻译，相应的语言的代码文件要放在跟模块文件相邻的目录下。从清单 5-35 中可以看到 `nls/de` 模块的代码。

清单 5-35 NLS 模块的组织模式



一旦特定语言的目录被创建，与默认语言模块相同名字的语言模块文件也要随之创建。新的语言模块会仅仅包含默认语言模块中 `root` 属性中被翻译的内容。清单 5-36 展示了 `pageTitle` 和 `searchPlaceholder` 两个属性的德语（`de`）翻译。

清单 5-36 德语翻译模块

```
// example-008/public/scripts/nls/de/lang.js
define({
  pageTitle: 'Grübeleien',
  searchPlaceholder: 'suche'
});
```

当默认的语言模块和 `i18n` 一起加载之后，它会检查浏览器的 `window.navigator.language` 属性，以确定应该使用什么样的语言环境和语言翻译。如果默认的语言模块指定了一个兼容且可用

的地区，`i18n` 会加载与该区域特指的模块，然后与默认的语言模块的 `root` 对象合并。若某个区域模块中缺少翻译，该模块就会被默认语言模块中的值填充。

图 5-3 所示的是当谷歌 Chrome 浏览器的语言已经被设置为德语后，名人名言页面的外观。

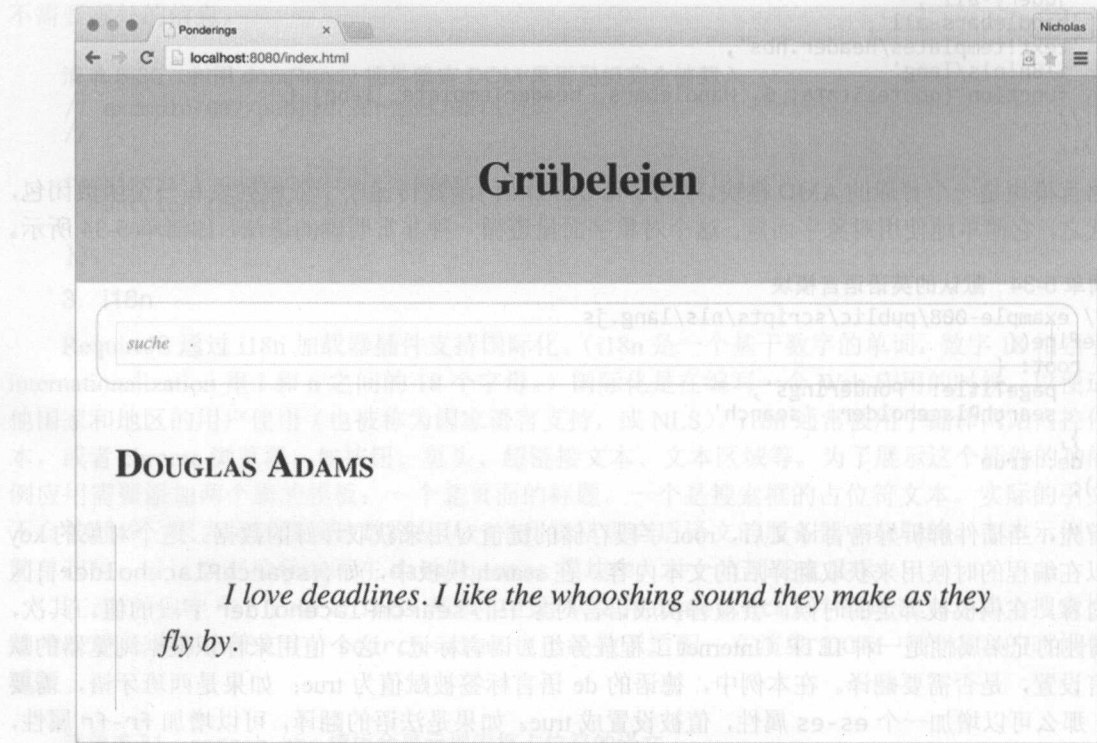


图 5-3 开启浏览器语言加载德语翻译选项

注意 `window.navigator.language` 在不同的浏览器中的设置是不同的。例如，在谷歌 Chrome 浏览器中，它只会被语言设置中的值修改；而在 Mozilla Firefox 浏览器中，它可以被页面的 HTTP 响应头的 `Accept-Language` 影响。

5.2.7 缓存清除

应用程序服务器经常在请求一个资源的时候发现与上次请求的内容并没有变化的时候，缓存资源，如脚本、图片、样式文件等，以消除不必要的磁盘访问。缓存的资源通常存储在内存中，并与一些 `key`（通常是资源的 URL）关联。

当给定的地址的多个请求发生在指定的缓存周期内，资源是使用 `key`（URL）从内存中取出的。这在生产环境中是一个显著的优势，但在开发环境或者测试环境中，若每次代码变更发生或者引入新的资源的时候，失效的缓存就会很令人讨厌。

当然，缓存可以在每个环境上打开或关闭，但更简单的解决方案是，可以至少为 JavaScript 资源（或者由 RequireJS 加载的任何资源）应用 RequireJS 的缓存无效化功能。缓存清除的过程是为每个资源请求改变其 URL 地址，以这种方式，资源仍可以被获取，但是不会在缓存中被发现，因为其“键”

总是不一样。通常是在页面重新载入的时候，改变一个查询字符串的时候完成该功能。

清单 5-37 的配置文件中增加了一个 `urlArgs` 属性，这将会在所有的文件发起请求之后增加一个由 RequireJS 生成的 `-bust={timestamp}` 的请求查询参数。时间戳是每次页面加载都会重新计算的，用来确保参数的改变，让 URL 变得唯一。

清单 5-37 `urlArgs` 配置可以用来做缓存清除

```
// example-009/public/scripts/main.js
requirejs.config({
  baseUrl: '/scripts',
  urlArgs: 'bust=' + (new Date().getTime()),
  paths: {
    // ...
  },
  shim: {
    // ...
  }
});
```

如图 5-4 所示，`bust` 参数的确是应用在每个 RequireJS 的请求中，甚至是 XHR 的请求——请求类似 `header.hbs` 这种文本资源的时候。

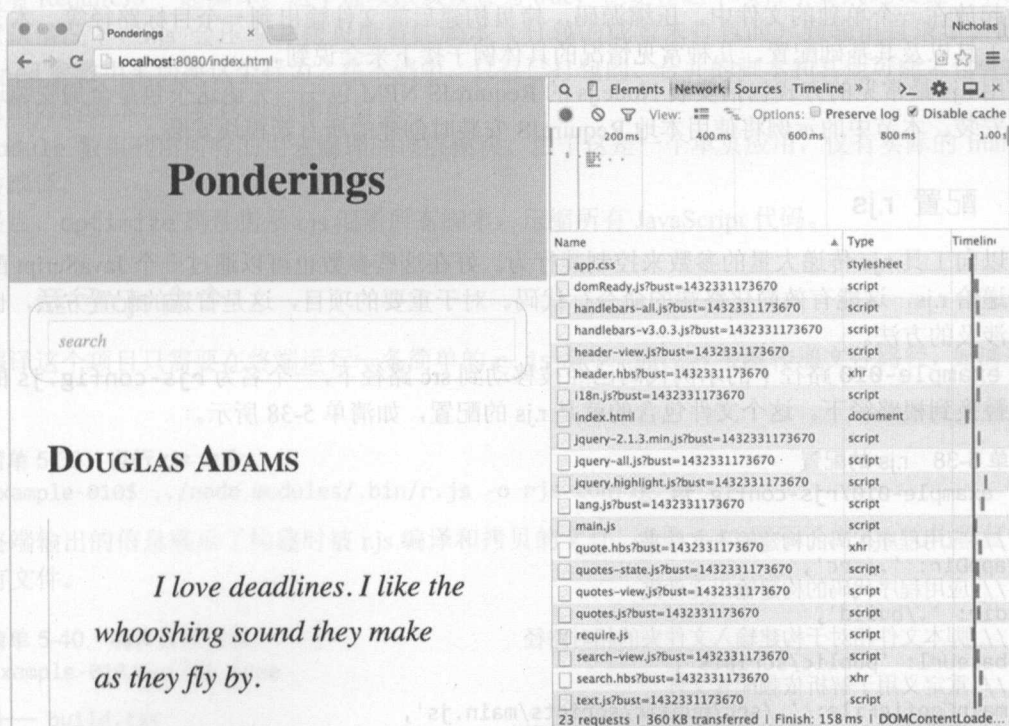


图 5-4 `bust` 参数已经添加到每个 RequireJS 请求之后

虽然此功能的实用性是显而易见的，但其仍会产生几个小问题。

首先，RequireJS 是遵守 HTTP 的缓存头设置的。所以，即使 `urlArgs` 被当作缓存清除使用了，RequireJS 可能仍然请求（或者接收）一个缓存版的资源，取决于如何设置缓存策略。如果可能，尽

可能在每个环境中设置不同的缓存头。

其次，要注意，某些代理服务器会丢掉查询参数，如果开发环境或者预上线环境包含一些代理去模拟生产环境，缓存清除的查询参数可能会失效。某些开发者用 `urlArgs` 参数在生产环境中指定特定的资源版本（如 `version=v2`）。但是通常不鼓励出于这个原因去使用，它是一个不可靠的版本技术——从最佳实践角度来说。

最后，一些浏览器把每个资源视为不同的 URL 并分别作为独立的可调试实体。在 Chrome 或者 Firefox 中，例如，如果一个调试断点被设置在 `http://localhost:8080/scripts/quotes-state.js?bust=1432504595280` 的源码中，当页面用 `http://localhost:8080/scripts/quotes-state.js?bust=1432504694566` 这个资源刷新的时候，断点会被移除。重置断点令人生厌。尽管 debugger 关键字可以通过强制浏览器暂停执行来避免这个问题，这仍然需要一个勤奋的开发人员确保生产环境中的代码已经去掉所有的 debugger 关键字。

5.3 RequireJS 优化

RequireJS 优化器 `r.js` 是用于 RequireJS 项目的构建工具。它可以用来把所有的 RequireJS 模块集中在一起放在一个单独的文件中，压缩源码，拷贝构建后的文件输出到一个目标路径等。本节介绍了这个工具以及其基础配置。几种常见情况的具体例子接下来会说到。

使用 `r.js` 最常见的方式包括安装 Node.js 的 RequireJS NPM 包——无论这个包是全局安装还是项目本地安装。本节中的示例将使用本地 RequireJS 安装时创建的所有新模块安装。

5.3.1 配置 `r.js`

可以向工具 `r.js` 传递大量的参数来控制其行为。好在这些参数也可以通过一个 JavaScript 配置文件来传递给 `r.js`，这能有效简化终端的命令行代码。对于重要的项目，这是首选的配置方法，也是本章唯一涉及的方法。

在 `example-010` 路径下的示例代码已经被移动到 `src` 路径下，一个名为 `rjs-config.js` 的文件已经被替换到根路径下。这个文件包含的就是 `r.js` 的配置，如清单 5-38 所示。

清单 5-38 `r.js` 的配置

```
// example-010/rjs-config.js
({
  // 应用程序代码的构建输入文件夹
  appDir: './src',
  // 应用程序代码的构建输出文件夹
  dir: './build',
  // 脚本文件相对于构建输入文件夹的相对路径
  baseUrl: 'public/scripts',
  // 重定义用于解析依赖配置文件
  mainConfigFile: './src/public/scripts/main.js',
  // 把所有对 text! 的引用以内联模块的形式引入
  inlineText: true,
  // 不需要拷贝构建后已合并的文件
  removeCombined: true,

  // 指定构建模块
  modules: [
    {
```



```

    name: 'main'
  },
],
// 混淆输出
optimize: 'uglify'
})

```

熟悉构建工具的开发人员会很熟悉配置中出现的输入/输出模式。

appDir 属性指定项目的“输入”目录，相对于配置文件，这里是编译的源代码所在的地方。

dir 属性指定项目的“输出”目录，相对于配置文件，在 **r.js** 工具运行时，被合并和压缩的文件会输出到这里。

baseUrl 属性告诉 **r.js** 项目的脚本相对于 **appDir** 属性值，其位置在哪儿。不能把它与 **main.js** 中的 **baseUrl** 属性搞混，**main.js** 中的 **baseUrl** 用来告诉 **RequireJS** 模块相对于 **Web** 应用的根目录的相对位置。

mainConfigFile 属性指向实际的 **RequireJS**（不是 **r.js**）配置，这有助于帮助 **r.js** 理解模块间的依赖关系，有哪些模块别名和 **shim** 的脚本存在——如果有的话。在 **r.js** 中可以省略该属性和指出的这些所有的路径——虽然这超出了这个例子的范围。

inlineText 属性设置为 **true**，确保所有引用的文本插件前缀 **text!** 的文本文件在最后编译输出的时候随着 **RequireJS** 一起编译。这个选项默认设置为 **true**，但是在这个项目中显式地设置一下。

默认情况下，**r.js** 会压缩并拷贝所有的脚本（打包的或者未打包的）到输出目录下。**Remove Combined** 参数用来切换这个行为。在本例中，只有已打包的、合并的脚本和其他可能未包含在打包输出范围内的脚本，会被拷贝到输出目录。

module 数组列出所有的用来编译的顶层模块。由于这是一个单页应用，仅有实际的 **main** 模块需要被编译。

最后，**optimize** 属性指示 **r.js** 混淆所有脚本，压缩所有 **JavaScript** 代码。

5.3.2 运行 r.js 命令

编译这个项目只需要在终端运行一条简单的 **r.js** 命令，通过 **-o** 参数传递文件路径，如清单 5-39 所示。

清单 5-39 运行 **r.js** 命令

```
example-010$ ../node_modules/.bin/r.js -o rjs-config.js
```

终端输出的信息展示了构建时被 **r.js** 编译和拷贝的文件。清单 5-40 展示了由 **r.js** 优化和复制输出的所有文件。

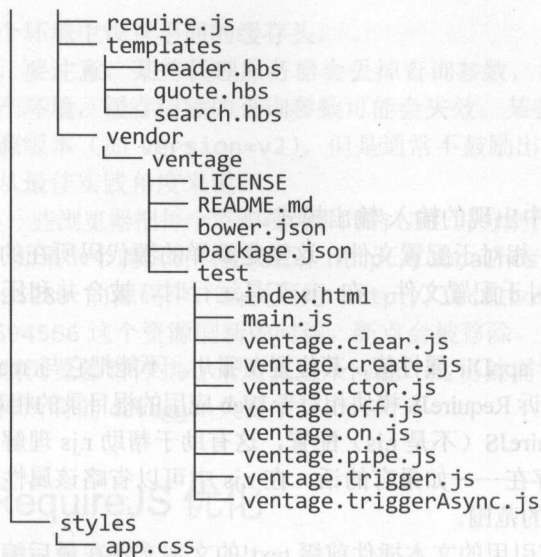
清单 5-40 编译目录内容

```
example-010/build$ tree
```

```

├── build.txt
├── index.js
├── public
│   ├── index.html
│   └── scripts
│       ├── main.js
│       ├── nls
│       └── de
│           └── lang.js

```



9 directories, 24 files

public/script 目录下清晰显示了以下几件事。

第一，`require.js` 和 `main.js` 的脚本都在顶层。由于这两个脚本是 `index.html` 中唯一引用的文件，这也是这两个脚本存在于这个位置的原因。其他脚本，比如 `quotes-view.js` 和 `quotes-state.js` 脚本很明显没被构建输出，但通过查看 `main.js` 文件的内容，你可以明白为什么：它们按照 `r.js` 的设置被打包和压缩在 `main.js` 中。

第二，本地文件 `nls/lang.js` 也不见了，因为它已经被包含到 `main.js` 中。`nls/de/lang.js` 脚本仍然作为编译输出的一部分，但它的内容已经被压缩了。任何一个浏览器访问这个页面，其默认的语言环境将得到优化后的体验，RequireJS 不会把默认的翻译作为一个 AJAX 请求来加载。来自德国的用户将会有额外的 HTTP 请求，因为德语本地化文件没有被列入打包输出。这是 `r.js` 必须遵守的本地化插件限制。

第三，Handlebars 模板尽管被编译输出到 `main.js`，也同样被拷贝到 `public/scripts/templates` 路径下。这是因为 RequireJS 插件目前还没有可视化构建过程，因此没有履行的 `r.js` 配置文件中的 `removeCombined` 选项的方法。幸运的是，因为这些模板已经被包装在 AMD 模块中和 `main.js` 连接在一起，RequireJS 不会尝试用 AJAX 请求加载它们。如果部署的大小是这个项目的瓶颈，可以创建一个构建后脚本或任务，删除该模板目录。

第四，`vendor/ventage` 目录已经被拷贝到 `build` 目录，尽管它的核心模块 `ventage.js` 已经被打包进了 `main.js`。虽然 RequireJS 编译后可自动删除单个模块文件（如 `ventage.js`），但它不会清理与模块相关的其他文件（在本例中，比如单元测试和包的定义文件，如 `package.json` 和 `bower.json`），所以它们必须被手动移除，或作为构建后过程的一部分。

5.4 小结

RequireJS 是一个非常实用的 JavaScript 模块加载器，在浏览器环境下运行良好。它的异步加载和解析模块的能力表明它对性能的优化不仅仅依赖于打包和压缩。为了更进一步优化，`r.js` 优化工具

可以用来把一些 RequireJS 模块打包成一个单独的、被压缩的脚本，以减少必要模块和资源的 HTTP 请求数量。

虽然 RequireJS 的模块必须是 AMD 格式定义，但 RequireJS 还可以使用被添加垫片的非 AMD 脚本，以便旧的代码可以通过 AMD 的模块被导入需要的位置。被 shim 的脚本也可以包含依赖项，因为该依赖项可以通过 RequireJS 自动加载。

text 插件可以让模块导入外部文本文件（如模板）为字符串。这些文本文件加载起来就像任何其他模块的依赖，甚至可能由 r.js 优化编译被内联输出。

i18n 模块加载器支持本地化，它可以动态加载基于浏览器的区域设置文本的翻译模块。本地翻译模块可以通过 r.js 打包和压缩，其他区域的翻译模块将始终以 HTTP 请求的形式加载。

模块的执行可以通过 pageLoad 插件延迟加载，避免模块的回调在 DOM 被完全渲染之前执行。这可以有效消除 jQuery 的 ready() 方法的重复调用，或消除手动订阅在 DOMContentLoaded 事件上的跨浏览器兼容代码。

最后，RequireJS 配置可以把查询参数自动追加到所有 RequireJS 的 HTTP 请求，给开发环境提供了成本低廉而又有效的缓存清除功能。

Browserify

少即是多。

——路德维希·密斯·凡德罗

Browserify 是一个 JavaScript 模块加载器，用来弥补当前浏览器不支持模块加载的缺陷，是一个对代码的预处理器。它在处理方式上与 CSS 的扩展大致相同。例如，SASS 和 LESS 对样式表提供了更强的语法，Browserify 通过调用全局的 `require()` 方法递归扫描源代码，增强了客户端的 JavaScript 应用。当 Browserify 发现这样的调用，它会立即加载被引用的模块（类似的 Node.js 可用 `require()` 函数），把多个模块合并为一个单独的、被压缩文件——一个可以被浏览器直接加载的“bundle”文件。

这种简单而优雅的方法将 CommonJS（将模块加载到 Node.js 中的方法）的功能和便利性带到浏览器端，也去掉了那些额外的复杂的样板代码，样板代码是异步模块定义（AMD）加载器如 RequireJS（第 5 章介绍的）所需要的。

在本章中，你会学到如何：

- 区分 AMD 与 CommonJS 模块加载器
- 通过工具创建模块化的客户端 JavaScript 应用，其模块化方式遵循由 Node.js 所普及的简单模式
- 可视化一个项目依赖树
- 使用 Browserify 的姐妹应用 Watchify，在项目代码变化时尽快重新编译代码
- 使用第三方 Browserify 插件（transform）为工具核心功能之外进行扩展

■ **注意** 部分本章讨论的内容已经在本书前面的第 1 章 Bower 和第 2 章 Grunt 提到过。如果你对 这些工具不熟悉，在继续之前，我们鼓励你去温习之前的内容。

6.1 AMD API 与 CommonJS 对比

第 5 章中涉及的 AMD 的 API 提供了一个很巧妙的方式来解决 JavaScript 目前缺乏的模块内联加载功能，通常被称为“浏览器优先”。AMD 的 API 通过让开发者把每一个模块用回调函数包装来实现模块载入浏览器，使模块随后可以异步加载（如“懒加载”）。这个过程被展示在清单 6-1 的模块中。

清单 6-1 AMD 模块的定义和引用

```
// requirejs-example/public/app/weather.js

define([], function() {
    return {
        'getForecast': function() {
            document.getElementById('forecast').innerHTML = 'Partly cloudy.';
        }
    };
});

// requirejs-example/public/app/index.js

define(['weather'], function(weather) {
    weather.getForecast();
});
```

AMD 的 API 做法既巧妙又有效，但许多开发人员还是发现它有点笨拙和冗长。理想情况下，JavaScript 应用程序应该能够在不增加复杂度及不引入 AMD API 所必需的样板代码情况下，引用外部模块。幸运的是，被称为 CommonJS 的流行的替代品解决了这个问题。

时下，大部分人倾向于把 JavaScript 与 Web 浏览器联系起来，事实是 JavaScript 在很长一段时期内被广泛使用在其他的一些环境中——当然是指在 Node.js 到来之前。这些环境包含 Rhino，由 Mozilla 创造的一个服务端的运行时环境，还有 ActionScript——一个被用于由 Adobe 开发的一度曾经火爆但是近些年已经不流行的 Flash 平台的派生脚本。这些平台为支持 JavaScript 对模块化的内置不足都创建了各自的解决办法。

一组开发人员意识到需要一个标准来解决这个问题，他们聚在一起，提出了被称为 CommonJS 的标准化的方法来定义和使用 JavaScript 模块。Node.js 遵循了类似的做法，下一个 JavaScript 重大的更新 (ECMAScript, ES6) 也将应用这个做法。这种方式也可以被用来书写模块化的 JavaScript 应用并且在今天所有的浏览器上使用——尽管这种方式不需要本章提到的 Browserify 这种额外的工具。

6.2 安装 Browserify

在开始之前，你应该确保已经安装 Browserify 的命令行工具。安装包是一个 npm 包，安装过程如清单 6-2 所示。

清单 6-2 通过 npm 安装 browserify 的命令行工具

```
$ npm install -g browserify
$ browserify --version
10.2.4
```

注意 Node 的包管理器 (npm) 允许用户把包安装到两种环境：本地或者全局。在这个例子中，browserify 被安装到全局的环境中，通常是为在命令行操作而保留的。

6.3 创建你的第一个 Bundle

Browserify 的大部分吸引力在于简单。熟悉 CommonJS 和 Node 的开发者会发现它用起来像立即

回到家一样自在。通过举例的方式，我们看一下清单 6-3。它展示了基于 CommonJS 的应用程序，与清单 6-1 中基于 RequireJS 的应用程序等价。

清单 6-3 通过 CommonJS 进行模块依赖的前端应用

```
// simple/public/app/index.js

var weather = require('./weather');
weather.getForecast();

// simple/public/app/weather.js

module.exports = {
  'getForecast': function() {
    document.getElementById('forecast').innerHTML = 'Partly cloudy.';
  }
};
```

与基于 RequireJS 的例子不同的是，该应用不能直接在浏览器中运行。这是由于浏览器的缺陷：没有一个内置的装置用来通过 `require()` 方法加载模块。在浏览器能运行应用之前，我们首先要把应用编译成一个包，可以使用 `browserify` 命令行工具或者使用 Browserify 的 API。利用 Browserify 的命令行工具来打包的命令如下。

```
$ browserify app/index.js -o public/dist/app.js
```

下面通过 `browserify` 打包我们应用的主文件 `public/app/index.js`，声明编译的输出应该被保存在 `public/dist/app.js`，这段编译后的脚本在项目的 HTML 中被引用（见清单 6-4）。

清单 6-4 HTML 引用了被 Browserify 编译后的 bundle 文件

```
// simple/public/index.html

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Browserify - Simple Example</title>
</head>
<body>
  <div id="forecast"></div>
  <script src="/dist/app.js"></script>
</body>
</html>
```

除了使用 Browserify 的命令行工具，我们也可以通过 Browserify 提供的 API 的方式来编写应用程序。这样做会很容易地让我们在一个更大的构建过程中插入这一步（如用 Grunt 工具开发）。清单 6-5 展示了这个项目名为 `browserify` 的 Grunt 任务。

清单 6-5 通过 Browserify 的 API 编译程序的 Grunt 任务

```
// simple/tasks/browserify.js

module.exports = function(grunt) {

  grunt.registerTask('browserify', function() {
    var done = this.async();
    var path = require('path');
```

```

var fs = require('fs');
var src = path.join('public', 'app', 'index.js');
var target = path.join('public', 'dist', 'app.js');
var browserify = require('browserify')([src]);
browserify.bundle(function(err, data) {
  if (err) return grunt.fail.fatal(err);
  grunt.file.mkdir(path.join('public', 'dist'));
  fs.writeFileSync(target, data);
  done();
});
});
};

```

6.4 可视化依赖树

如果碰巧你是一个更需要视觉表现力的学习者，对于如何表达 Browserify 编译过程中所发生的事情，图 6-1 所示的图表会大有帮助。在这里，我们看到的是本章中 Browserify 编译 advanced 项目的各种依赖关系的可视化。

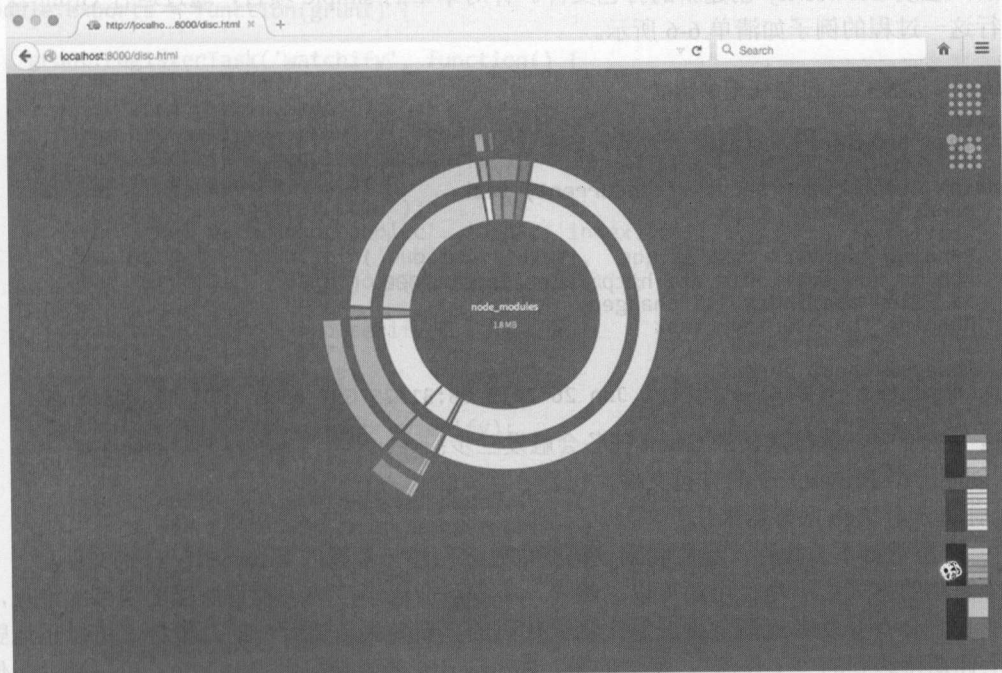


图 6-1 advanced 项目的可视化依赖树

把一个编译过程静态化成一个图表来说，是不太公平的。为查看完整的效果，你应该在项目目录下通过 `npm start` 命令来编译你的项目，并在浏览器中查看图表。这样做能让你在鼠标滑动到图表上的时候，显示其对应的依赖。每一项都对应其编译过程中发生的依赖。虽然在图 6-1 中并不明显，但深入图表分析表明，我们应用程序自定义的代码只占被 Browserify 打包之后生成的文件的很小一部分（9.7 KB），作为项目主体的接近 2 MB 大小的代码由第三方依赖（如 Angular、jQuery、Lodash

等) 构成。本章的后半部分会再次提到这个重要的事实。

■ **注意** 也许你对研究 `browserify-graph` 和 `colony` 命令行工具 (同样可以用 `npm` 安装) 也感兴趣, 你可以使用它们生成额外的可视化的项目依赖树。

6.5 发生变化时重新打包文件

使用了 `Browserify` 的项目不能直接运行在浏览器, 必须先编译。为了最有效地利用工具, 在源代码发生变化的时候自动触发这一步, 对建立项目来说是非常重要的。下面看看可以实现的两种方法。

6.5.1 通过 Grunt 监听文件变化

从第2章讲的 `Grunt` 内容中, 你会发现, 插件如 `grunt-contrib-watch` 之类允许开发者在他们的项目代码变化之后触发编译步骤。很容易看到这样的工具是如何应用到项目中的。当监测到新的变化之后, 触发 `Browserify` 创建新的打包文件。针对本章的 `simple` 项目, 通过运行 `Grunt` 默认的任务来执行这一过程的例子如清单 6-6 所示。

清单 6-6 通过 Grunt 触发新的构建

```
$ grunt
Running "browserify" task

Running "concurrent:serve" (concurrent) task
  Running "watch" task
    Waiting...
  Running "server" task
    App is now available at: http://localhost:7000
  >> File "app/index.js" changed.
  Running "browserify" task

Done, without errors.
Completed in 0.615s at Fri Jun 26 2015 08:31:25 GMT-0500 (CDT) - Waiting...
```

在这个例子中, 运行默认的 `Grunt` 任务会触发三步:

- 创建一个 `Browserify` 的打包文件。
- 启动一个 `Web` 服务器来托管项目。
- 一个监听脚本被执行, 当监测到代码变化时, 创建一个新的 `Browserify` 打包文件。

这个简单的方式给小项目用还可以; 然而, 小项目会逐渐变成大项目, 伴随项目的增长, 构建时间也变长, 会令开发者逐渐沮丧。每次测试代码的更新效果需要等待的几秒钟, 可以迅速毁掉任何意义上你期待达到的“工作流”。幸运的是, `Browserify` 的姐妹应用 `Watchify`, 可以帮助我们解决这些状况。

6.5.2 通过 Watchify 监听文件变化

如果把 `Browserify` (其编译了整个应用程序) 比喻为一个切肉刀, `Watchify` 则可以被比喻为削皮刀。当被调用时, 初始时 `Watchify` 编译的是整个项目; 但是, `Watchify` 并不是一旦此过程完成就退出, 而是会继续运行, 继续监测项目的源代码变更。检测到更改后, `Watchify` 只会

编译那些已更改的文件，从而大幅度提高构建的时间。Watchify 通过自己内部的缓存机制来保持每次构建。

与 Browserify 一样，可以通过命令行调用 Watchify 或其提供的 API。在清单 6-7 中，本章的 `simple` 项目是通过 Watchify 的命令行工具编译的。在本例中，`-v` 参数传递给命令行，用来指定 Watchify 在详细模式下运行。因此，Watchify 会告诉我们变化已经检测到。

清单 6-7 通过 npm 安装 Watchify，并在本章的 `simple` 项目中运行

```
$ npm install -g watchify
$ watchify public/app/index.js -o public/dist/app.js -v
778 bytes written to public/dist/app.js (0.03 seconds)
786 bytes written to public/dist/app.js (0.01 seconds)
```

与 Browserify 一样，Watchify 提供了一个方便的 API，允许我们将其集成到一个更大的构建过程（见清单 6-8）。我们做的只是一点小改动，如清单 6-7 所示的 Browserify 任务。

清单 6-8 通过 Grunt 任务演示 Watchify 的 API 的使用

```
// simple/tasks/watchify.js
```

```
module.exports = function(grunt) {
  grunt.registerTask('watchify', function() {
    var done = this.async();
    var browserify = require('browserify');
    var watchify = require('watchify');
    var fs = require('fs');
    var path = require('path');
    var src = path.join('public', 'app', 'index.js');
    var target = path.join('public', 'dist', 'app.js');
    var targetDir = path.join('public', 'dist');

    var browserify = browserify({
      'cache': {},
      'packageCache': {}
    });
    browserify = watchify(browserify);
    browserify.add(src);

    var compile = function(err, data) {
      if (err) return grunt.log.error(err);
      if (!data) return grunt.log.error('No data');
      grunt.file.mkdir(targetDir);
      fs.writeFileSync(target, data);
    };

    browserify.bundle(compile);

    browserify.on('update', function() {
      browserify.bundle(compile);
    });

    browserify.on('log', function(msg) {
      grunt.log.oklns(msg);
    });
  });
};
```

在这个例子中，我们用 `watchify` 包裹了 `browserify` 实例，然后通过订阅的 `update` 事件触发被包裹的实例来完成项目的重新编译。

6.6 使用多个打包文件

在前面的“可视化依赖树”部分，我们看到了一个交互式图表。在 `Browserify` 编译这一章的 `advanced` 项目（见图 6-1）中，我们可以把遇到的各种依赖关系可视化。最重要的一个事实是，我们可以从这个图中发现，项目的自定义代码（见 `/app` 目录）在整个打包代码的 1.8 MB 大小中只占了小小的一部分（9.7 KB）。换句话说，这个项目的绝大多数的代码是包含的第三方库（如 `Angular`、`jQuery`、`Lodash` 等），不太可能经常变。让我们看看如何把这一知识点更有效地利用。

本章的 `extracted` 项目与 `advanced` 项目从各方面来说是相同的，只有一个不同：跟只编译了一个 `Browserify` 打包文件不同，`extracted` 构建过程创建了两个打包文件：

- `/dist/vendor.js`：第三方依赖
- `/dist/app.js`：自定义应用代码

通过这种方式，浏览器可以更好地访问项目发布的更新。换句话说，变化只体现在项目的自定义代码中，浏览器只需要重新下载 `/dist/app.js`。对比 `advanced` 项目，每次更新（不管有多少）都要强制重新下载接近 2 MB 大小的包文件。

清单 6-9 展示了 `extracted` 项目的 HTML 文件。如你所见，这里引入了两个分开的包文件：`/dist/vendor.js` 和 `/dist/app.js`。

清单 6-9 本章 `extracted` 项目的 HTML 文件

// `extracted/public/index.html`

```
<!DOCTYPE html>
<html ng-app="app">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Browserify - Advanced Example</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body class="container">

    <navbar ng-if="user_id"></navbar>

    <div ng-view></div>

    <footer><a href="/disc.html">View this project's dependency tree</a></footer>

    <script src="/dist/vendor.js"></script>
    <script src="/dist/app.js"></script>

  </body>
</html>
```

清单 6-10 展示了 `extracted` 项目的 `Gruntfile`。注意，示例中设置了一个特殊的值（`browserify.vendor_modules`）。

清单 6-10 本章 extracted 项目的 Gruntfile

```
// extracted/Gruntfile.js
```

```
module.exports = function(grunt) {
```

```
  grunt.initConfig({
    'browserify': {
      'vendor_modules': [
        'angular',
        'bootstrap-sass',
        'jquery',
        'angular-route',
        'angular-sanitize',
        'restangular',
        'jquery.cookie',
        'lodash',
        'underscore.string',
        'lodash-deep'
      ]
    }
  });
```

```
  grunt.loadTasks('tasks');
```

```
  grunt.registerTask('default', ['compass', 'browserify', 'browserify-vendor',
    'init-db', 'concurrent']);
};
```

清单 6-11 展示了 extracted 项目的 browserify 的 Grunt 任务。这个任务主要模拟在 advanced 项目中的任务。不同的是，本任务迭代循环了 Gruntfile 中定义的第三方包。针对每一个条目，我们会引导 Browserify 在编译的包文件中排除其引用。

清单 6-11 extracted 项目的 browserifyGrunt 任务

```
// extracted/tasks/browserify.js
```

```
module.exports = function(grunt) {
```

```
  grunt.registerTask('browserify', function() {
```

```
    var done = this.async();
    var path = require('path');
    var fs = require('fs');
    var target = path.join('public', 'dist', 'app.js');
    var vendorModules = grunt.config.get('browserify.vendor_modules') || [];
    var browserify = require('browserify')([
      path.join('app', 'index.js')
    ], {
      'paths': ['app'],
      'fullPaths': true,
      'bundleExternal': true
    });
```

```
    vendorModules.forEach(function(vm) {
      grunt.log.writeln('Excluding module from application bundle: %s', vm);
      browserify.exclude(vm);
    });
```

```
    browserify.bundle(function(err, data) {
      if (err) return grunt.fail.fatal(err);
      grunt.file.mkdir(path.join('public', 'dist'));
      fs.writeFileSync(target, data);
    });
```

```

        grunt.task.run('disc');
        done();
    });
};

```

最后, 清单 6-12 展示了 **extracted** 项目中 Grunt 任务 **browserify-vendor** 的内容。运行该任务后, Browserify 会构建产出一个单独的包, 其仅由清单 6-10 中定义的第三方模块组成。

清单 6-12 extracted 项目中的 Grunt 任务 browserify-vendor

```
// extracted/tasks/browserify-vendor.js
```

```

module.exports = function(grunt) {

    grunt.registerTask('browserify-vendor', function() {

        var done = this.async();
        var path = require('path');
        var fs = require('fs');
        var target = path.join('public', 'dist', 'vendor.js');
        var vendorModules = grunt.config.get('browserify.vendor_modules') || [];

        var browserify = require('browserify')({
            'paths': [
                'app'
            ],
            'fullPaths': true
        });
        vendorModules.forEach(function(vm) {
            browserify.require(vm);
        });

        browserify.bundle(function(err, data) {
            if (err) return grunt.fail.fatal(err);
            grunt.file.mkdir(path.join('public', 'dist'));
            fs.writeFileSync(target, data);
            done();
        });
    });
};

```

为看到这个过程, 在终端中切换到 **extracted** 项目, 然后运行 **\$ npm start**。任何缺失的 npm 模块将被安装, 项目的默认 Grunt 任务也会被执行。这个过程执行后会创建两个单独的打包文件。其中, 包含项目自定义代码的打包文件 **/dist/app.js**, 只有 14 KB 大小。

6.7 Node 方式

正如本章所提到的, Browserify 编译一个项目的方式是递归扫描源码中全局 **require()** 函数。随着函数调用被一个个找到, Browserify 会用和 Node 一样的 **require()** 方法载入这些被引用的模块。然后, Browserify 把它们合并成一个浏览器能理解的打包文件。

就这一点而言, Browserify 最好是用于客户端的 Node 项目。Browserify 在许多方面会给新手造成困惑, 但你只要牢记这个概念就会觉得它更容易理解: 关联它包含的一切。现在我们看以下两个

方面：模块解析方案和依赖关系管理。

6.7.1 模块解析方案和 NODE_PATH 环境变量

Node 应用程序可以用多种方式引用模块。例如，这里我们看到一个简单 Node 应用依赖了一个相对路径的模块：

```
var animals = require('./lib/animals');
```

按相似方式，该示例也可以引用完整绝对路径的模块。无论哪种方式，很明显 Node 都希望能找到这个模块。现在思考下面的例子，仅通过名字引用模块：

```
var animals = require('animals');
```

在这样的情况下，Node 将首先尝试在其核心库中查找引用的模块。在实际中，你可以看到这样的过程，如加载 Node 的 `fs` 模块时。如果没有找到匹配，Node 将继续在一个名为 `node_modules` 的文件夹下，通过文件系统搜索被 `require()` 方法调用的模块。一旦遇到匹配的文件夹，Node 将检查该文件夹内是否包含与请求相匹配的模块（或包）。这个过程会继续，直到找到一个匹配模块；如果没有找到，则抛出异常。

这个 Node 模块解析方案尽管简单，但功能强大，其几乎仅仅围绕着 `node_modules` 文件夹来处理。然而，Node 提供了一个常常被忽视的方法，它允许开发者定义一个额外的目录来让 Node 去查找模块，应该在前面的方式无效时增加这个步骤。让我们看一看这一章的 `path-env` 项目是如何做到的。

清单 6-13 展示了项目中 `package.json` 文件中的一段代码。特别重要的是已经定义的 `start` 脚本。根据已经展示的配置，当在项目中运行 `$ npm start` 的时候，在应用运行之前，`NODE_PATH` 环境变量将被更新成包含了项目的 `/lib` 目录。其结果是，Node 会把此文件夹加入到命名模块解析的查找范围内。

清单 6-13 这个项目的 `npm start` 脚本更新 `NODE_PATH` 环境变量

```
// path-env/package.json
```

```
{
  "name": "path-env",
  "version": "1.0.0",
  "main": "./bin/index.js",
  "scripts": {
    "start": "export NODE_PATH=$NODE_PATH:./lib && node ./bin/index.js"
  }
}
```

■ **注意** 在 OS X 和 Linux 系统中，环境变量的设置是通过在终端中运行 `export ENVIRONMENT_VARIABLE=value`。在 Windows 操作系统中，其命令是 `set ENVIRONMENT_VARIABLE=value`。

设置 `NODE_PATH` 环境变量的重要性可能并不明显；然而，这样做可以显著提高项目的整洁度和可维护性。因为使用这种方法时，它本质上允许开发者创建一个命名空间，应用程序的模块（那些不作为独立的 `npm` 包存在的模块）可以通过名称而不是冗长的相对路径被引用。下面的简单示例介绍了实践中如何操作（见清单 6-14）。

清单 6-14 path-env 项目包含的几个模块

```
// path-env/bin/index.js

var api = require('app/api');

// path-env/lib/app/api/index.js

var express = require('express');
var path = require('path');
var app = express();
var animals = require('app/models/animal');
app.use('/', express.static(path.join(__dirname, '..', '..', '..', 'public')));
app.get('/animals', function(req, res, next) {
  res.send(animals);
});
app.listen(7000, function() {
  console.log('App is now available at: http://localhost:7000');
});
module.exports = app;

// path-env/lib/app/models/animal/index.js

module.exports = [
  'Aardvarks', 'Cats', 'Dogs', 'Lemurs', 'Three-Toed Sloths', 'Zebras'
];
```

注意，这个例子缺乏相对路径的模块引用。例如，项目的入口文件 `bin/index.js` 是可以通过 `require('app/api');` 载入一个负责初始化 Express 的模块。传统的方式是使用相对路径 `require('../lib/app/api');`。任何通过 `require('../../../../models/animal');` 这种冗长的方式开发复杂 Node 应用的人，都会认同这种清晰的写法所带来的益处。

注意 需要牢记的是，`NODE_PATH` 环境变量只在 Node（或者 Browserify）应用的上下文中有用——在一个包里是不好用的。在创建一个用于允许他人复用的包时，应仅依赖 Node 的默认模块解析方式。

在 Browserify 中使用 NODE_PATH

到目前为止，我们已经集中讨论了 `NODE_PATH` 环境变量如何助益于服务器端 Node 应用。我们已经奠定了基础，现在让我们来看看如何把这个概念应用于客户端，用 Browserify 编译基浏览器端应用。

清单 6-15 展示了本章 `advanced` 项目中的 Grunt 任务 `browserify`，它通过 Browserify 的 API 编译这个应用。特别重要的是使用 `paths` 选项，它可以让给 Browserify 提供一个路径数组，该路径需要在编译开始前放到 `NODE_PATH` 环境变量之后。通过这个设置，我们可以轻易地利用本节之前示例中提到的优点。

清单 6-15 本章的 `advanced` 项目的 Grunt 任务 `browserify`

```
// advanced/tasks/browserify.js

module.exports = function(grunt) {

  grunt.registerTask('browserify', function() {
    var done = this.async();
```

```

var path = require('path');
var fs = require('fs');
var target = path.join('public', 'dist', 'app.js');
var browserify = require('browserify')([
  path.join('app', 'index.js')
], {
  'paths': [
    'app'
  ],
  'fullPaths': true
});
browserify.bundle(function(err, data) {
  if (err) return grunt.fail.fatal(err);
  grunt.file.mkdir(path.join('public', 'dist'));
  fs.writeFileSync(target, data);
  grunt.task.run('disc');
  done();
});
});
};

```

思考一下清单 6-16 中的示例，思考它是如何使用这种方法助益项目的。我们可以看到一个小模块，负责加载 `Lodash` 并整合两个第三方工具 `underscore.string` 及 `lodash-deep`。最后，输出值是一个包含三个模块功能的单独对象。

清单 6-16 负责加载 `Lodash` 和集成各种第三方插件的模块

```
// advanced/app/utls/index.js
```

```

var _ = require('lodash');
_.mixin(require('underscore.string'));
_.mixin(require('lodash-deep'));
module.exports = _;

```

由于 `paths` 的值已经提供给 `Browserify`，我们的应用可以在任何一个路径下通过调用 `require('app/utls')`；来引用这个模块。

6.7.2 依赖管理

直到最近，“依赖关系管理”对（大多数）基于浏览器的客户端项目来说还是一个陌生的概念。然而潮流迅速转变，在很大程度上要感谢迅速普及的 `Node`，以及已经被本书提到过的额外工具（比如 `Bower`、`Grunt` 和 `Yeoman` 等）。对于曾经是（并在很大程度上仍然是）未开辟的“蛮荒的美国西部”的客户端开发来说，这些实用程序带来了急需的工具和指导。

对于解决依赖管理问题，`Bower` 已经通过提供给客户端开发者一个易于使用的机制来管理大量程序依赖的第三方库。对于刚接触这些概念的并且没使用过类似编译器如 `Browserify` 的开发人员来说，始终可以选择 `Bower` 来管理项目依赖。然而，当开发人员体验过 `Browserify` 工具的优势后，也就逐渐会放弃 `Bower` 了。

在本节的开始提到，作为客户端应用程序，项目采用 `Browserify` 是最好的思想。对于依赖管理这一说法的确重要。回想一下 `Browserify` 的编译过程，它扫描项目源代码中全局 `require()` 函数的调用。每当找到一个 `require()` 函数，这些调用会在 `Node` 中被执行，返回的是一个可以被客户端应用使用的值。这里重点强调，当项目的依赖仅仅通过 `npm`、`Node` 包管理器时，使用 `Browserify` 能大大简化项

目依赖管理。虽然从技术上讲，可以告诉 Browserify 如何从 Bower 中安装包，但通常情况下，它的麻烦要大于它的价值。

6.8 定义浏览器指定模块

考虑这样一个场景：在该场景中，你希望创建一个新模块，打算发布和通过 npm 分享。你想要这个模块既能在 Node 中也能在浏览器（通过 Browserify）中运行。为帮助实现这种方式，Browserify 支持使用 browser 来设置项目的 package.json 文件。当定义该属性之后，开发者就可以覆盖默认的位置来定位一个特定的模块。为了更好地理解工作过程，让我们看看两个简单的例子。

清单 6-17 展示了一个简单包的内容。这个包中存在两个模块：lib/node.js 和 lib/browser.js。根据包的 package.json 文件可知，其 main 模块是 lib/node.js。换句话说，当这个包在 Node 应用中被通过名称引用时，这个模块就会被 Node 加载。注意，一个额外的配置文件已经被定义：“browser”：“./lib/browser.js”。这个设置的结果就是，Browserify 会加载这个模块而不是 main 属性对应的模块。

清单 6-17 模块在两个不同的切入点：一个是 Node，一个是 Browserify

```
// browser1/package.json
```

```
{
  "name": "browser1",
  "version": "1.0.0",
  "main": "./lib/node.js",
  "browser": "./lib/browser.js"
}
```

```
// browser1/lib/browser.js
```

```
module.exports = {
  'run': function() {
    console.log('I am running within a browser.');
```

```
};
```

```
// browser1/lib/node.js
```

```
module.exports = {
  'run': function() {
    console.log('I am running within Node.');
```

```
};
```

你接下来将要看到，Browserify 的 browser 配置不会仅局限于简单地覆盖包的 main 模块位置，还可以用于覆盖多个模块路径。思考一下清单 6-18，通过例子中的方式，我们可以声明多个浏览器特指模块来覆盖相应路径，其中 package.json 文件中 browser 设置项为对象而不是字符串。

清单 6-18 为 Node 和 Browserify 展示多个特指的模块

```
// browser2/package.json
```

```
{
  "name": "browser2",
```



```

"version": "1.0.0",
"main": "./lib/node.js",
"browser": {
  "./lib/node.js": "./lib/browser.js",
  "./lib/extra.js": "./lib/extra-browser.js"
}
}

```

如清单 6-17 所示，按这种模式实现的模块会暴露自身的入口点：一个用于 Node，一个是用于 Browserify 编译的单独的应用。然而，本例更进一步解释了这个概念。当这个模块被编译后，它会一直尝试加载位于 `lib/extra.js` 的模块，而不是加载位于 `lib/extra-browser` 的模块。通过这种方式，`browser` 设置让我们能够非常方便地创建模块——无论是在 Node 中运行还是在浏览器中运行的模块。

6.9 用 Transforms 扩展 Browserify

开发者可以通过创建插件来做基于 Browserify 核心功能之上的构建，称为**转换**，利用发生的编译过程创建新的包。这类转换的包可以通过 npm 来安装，在 `package.json` 文件中通过 `browserify.transform` 包含的一组名称来启用。让我们来看几个使用的例子。

6.9.1 brfs

`brfs` 转换简化了加载文件内联内容的过程。它扩展了 Browserify 的编译过程，以搜索 `fs.readFileSync()` 方法的调用。每当发现该方法被调用，引用的文件会立即被加载并返回。

清单 6-19 展示了本章 `transforms-brfs` 项目中 `package.json` 文件中的一段。在这个例子中，`brfs` 模块已经被安装，并且被包含进 `browserify.transform` 配置的设置中。

清单 6-19 本章 `transforms-brfs` 项目中 `package.json` 文件的一段摘要

```
// transforms-brfs/package.json
```

```

{
  "name": "transforms-brfs",
  "dependencies": {
    "browserify": "^10.2.4",
    "brfs": "^1.4.0"
  },
  "browserify": {
    "transform": [
      "brfs"
    ]
  }
}

```

清单 6-20 展示了项目中 `/app/index.js` 模块的内容。在本例中，`brfs` 转换会加载 `/app/templates/lorem.html` 的内容，随后会被赋值给 `tpl` 变量。

清单 6-20 通过 `fs.readFileSync()` 加载模板

```
// transforms-brfs/app/index.js
```

```

var fs = require('fs');
var $ = require('jquery');

```

```
var tpl = fs.readFileSync(__dirname + '/templates/lorem.html', 'utf8');
$('#container').html(tpl);
```

6.9.2 folderify

跟 `brfs` 转换很像, `folderify` 转换允许你内联加载内容。与其一次操作一个文件, `folderify` 允许你快速加载多个文件的内容。思考以下例子, 清单 6-21 展示了本章 `transforms- folderify` 应用的内容。

清单 6-21 用 `folderify` 加载多个文件内容

```
// transforms-folderify/app/index.js
```

```
var $ = require('jquery');
var includeFolder = require('include-folder');
var folder = includeFolder(__dirname + '/templates');

for (var k in folder) {
  $('#container').append('<p>' + k + ': ' + folder[k] + '</p>');
}
```

正如前面的例子, 这个项目中的 `package.json` 文件已经被修改, 其 `browserify.transform` 数组包含 `folderify`。当编译的时候, `Browserify` 会搜索到包含 `include-folder` 的模块。当函数返回值被调用, `Browserify` 会载入这个指定目录下的所有文件内容, 然后以对象的形式返回。

6.9.3 bulkify

通过 `bulkify` 转换, 开发者可以在一次调用中导入多个模块。为了更好地了解它是如何工作的, 清单 6-22 展示了本章 `transforms-bulkify` 项目的主应用文件的片段内容。

清单 6-22 本章 `transforms-bulkify` 项目主应用文件

```
// transforms-bulkify/app/index.js
```

```
var bulk = require('bulk-require');
```

```
var app = angular.module('app', [
  'ngRoute'
]);
```

```
var routes = bulk(__dirname, [
  'routes/**/*.route.js'
]).routes;
```

```
app.config(function($routeProvider) {
```

```
  var defaultRoute = 'dashboard';
```

```
  .each(routes, function(route, route_name) {
    route = route.route;
    route.config.resolve = route.config.resolve || {};
    $routeProvider.when(route.route, route.config);
  });
```

```
  $routeProvider.otherwise({
    'redirectTo': defaultRoute
  });
});
```

```
});
```

```
});
```

这个例子演示了 Angular 应用环境中 Browserify 的使用方式。如果你对 Angular（在第 8 章中讲过）不熟悉，不要着急——这个例子中最重要的部分是 `bulk()` 方法的使用方式，这个方法使我们能够用 `require()` 多个模块来匹配一个或多个指定的模式（在本例中，指的是 `routes/**/*.js`）。

图 6-2 展示了这个项目的文件模式。正如你看到的，`app/routes` 模块包含三个文件夹，每个文件夹都展示了 Angular 应用中的一个路由。`bulkify` 转换让我们能通过单次调用 `bulk()`，快速的 `require()` 每一个模块。随后，我们就可以遍历结果对象，然后把每个路由传递给 Angular。

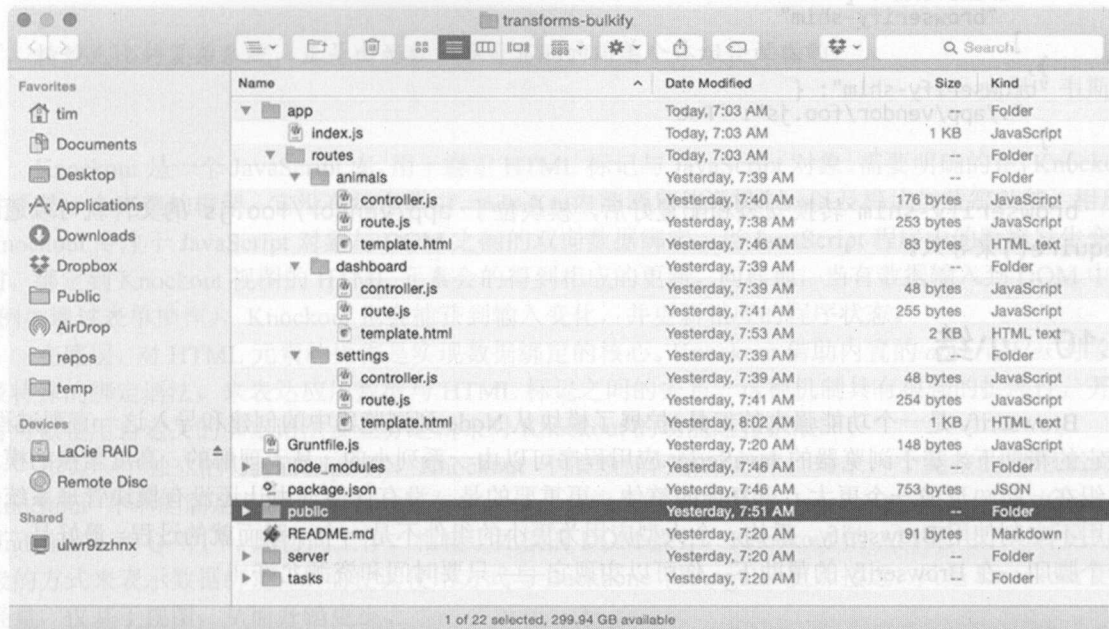


图 6-2 本章 transforms-bulkify 项目的文件模式

6.9.4 Browserify-Shim

开发人员使用 Browserify 时，偶尔会发现自己需要导入的模块不符合 CommonJS 规范。考虑下一个第三方模块 `Foo`，一旦它被加载，就会被分配给全局 `window.Foo` 变量（见清单 6-23）。这种类库可以通过 `browserify-shim` 转换的帮助来导入。

清单 6-23 第三方 `Foo` 类库，声明了一个全局 `Foo` 变量

```
// transforms-shim/app/vendor/foo.js
```

```
function Foo() {
  console.log('Bar');
}
```

正如之前清单 6-19 所示，通过 `npm` 在本地安装 `browserify-shim` 之后，通过在 `package.json`

文件中向启用转换数组添加该模块名来启用该转换。接下来在 `package.json` 文件中的第一级别创建一个 `browserify-shim` 对象，用作转换的配置（见清单 6-24）。在本例中，这个对象的每个键相当于通往实际暴露模块位置的路径，而相应的值指定了模块被赋予的全局的变量。

清单 6-24 项目的 `package.json` 文件中配置 `browserify-shim`

```
// transforms-shim/package.json
```

```
{
  "name": "transforms-shim",
  "version": "1.0.0",
  "main": "server.js",
  "browserify": {
    "transform": [
      "browserify-shim"
    ]
  },
  "browserify-shim": {
    "./app/vendor/foo.js": "Foo"
  }
}
```

`browserify-shim` 转换安装和配置好后，模块位于 `app/vendor/foo.js` 的文件就可以通过 `require()` 来导入。

6.10 小结

`Browserify` 是一个功能强大的工具，扩展了模块从 `Node` 到浏览器中的创建和导入这一直观过程。在它的帮助下，基于浏览器的 `JavaScript` 应用程序可以由一系列小的、易于理解的、高度聚焦的模块组织在一起，形成一个更大、更复杂的整体。更重要的是，没有任何理由让还没有模块管理系统的应用不立刻使用 `Browserify`。重构一个大型应用为更小的组件不是一个一蹴而就的过程，最好是一个脚印。在 `Browserify` 的帮助下，你可以实现它——只要时间和资源允许。

6.11 相关资源

- `Browserify`: <http://browserify.org>
- `Browserify transforms`: <https://github.com/substack/node-browserify/wiki/list-of-transforms>
- `brfs`: <https://github.com/substack/brfs>
- `Watchify`: <https://github.com/substack/watchify>

Knockout

精妙无比的复杂系统，正是由无数个各司其职的简单个体组合诞生的。

——大卫·韦斯特

Knockout 是一个 JavaScript 库，用于绑定 HTML 标记与 JavaScript 对象。需要明确的是，Knockout 并不是一个完整的框架，它没有状态路由、AJAX、内部消息传递机制，以及模块加载等功能。相反，Knockout 专注于 JavaScript 对象与 DOM 之间的双向数据绑定。当 JavaScript 程序中的数据发生变化时，绑定到 Knockout 视图的 HTML 元素会得到相应的更新。同样地，当有数据输入到 DOM 中时（例如通过表单操作），Knockout 也会捕获到输入变化，并更新相应的程序状态。

在底层，对 HTML 元素的操作是实现数据绑定的核心。Knockout 借助内置的 *observables* 对象以及特殊的绑定语法，来表达应用数据与 HTML 标记之间的关系。这种机制具有很强的扩展性，开发者可以使用自定义的绑定语法及业务逻辑来对 Knockout 的功能进行扩展。

作为一个独立的 JavaScript 库，Knockout 不依赖任何第三方库。对于一个完整的应用来说，Knockout 并不能满足其所有需求。好在 Knockout 能与很多常用的 JavaScript 库，例如 jQuery、Underscore 和 Q 等，一起协同工作。相较于死板的 DOM 操作，Knockout 的 API 采用了一种更为高级的方式来表示数据绑定，从而使其使用方式与 Backbone 和 Angular 更加接近，只是 Knockout 更为轻量，仅基于视图，从而开销更小。

Knockout 支持所有现代浏览器。截至本书撰写之时，Firefox 3+、Internet Explorer 6+以及 Safari 6+ 仍然受到支持。Knockout 有一项新特性，利用自定义标签实现了 HTML 兼容的组件。有鉴于此，其向后兼容性更加令人印象深刻。Knockout 的开发团队付出了很多努力，以确保在不同的浏览器中获得几乎相同的开发体验。

本章通过一个厨房菜谱管理程序，来探讨 Knockout 的特性与用法。所有的示例代码的前面都有一段注释，标明示例代码的文件路径。以清单 7-1 为例，文件 `index.js` 可以在本书源代码中的 `knockout/example-000` 目录下找到。

清单 7-1 一个没有任何实际作用的示例

```
// example-000/index.js
console.log('this is not a real example');
```

运行本例之前，请先安装 Node.js（具体安装过程请参考 Node.js 的官方文档），然后在 Knockout 目录下运行 `npm install`，以安装示例所依赖的第三方包。每个例子的目录下都会有一个 `index.js` 文件，该文件运行之后，会启动一个 Node.js 服务器。然后，我们就可以在浏览器中访问指定的 URL，

以查看程序运行的结果了。例如，要运行清单 7-1 中的示例，先在终端中切换到 `knockout/example-000` 目录，再执行 `node index.js` 命令即可。

本章几乎所有的示例页面都是通过 `<script>` 标签引入 Knockout 的核心脚本（core script）的。你可以从 <http://knockoutjs.com> 或者某个提供 Knockout 下载的 CDN 获取 Knockout 的脚本文件，也可以通过 Bower 或者 npm 来安装。Knockout 同时兼容 AMD 与 CommonJS 规范。关于上述几种安装方式的详细说明，请参考 Knockout 的官方文档。

7.1 View、Model 与 View Model

Knockout 将应用程序 UI 层的数据来源分为两种：数据模型（data model），代表程序的状态，以及视图模型（view model），决定该状态如何显示或者说传达给用户。在 Knockout 应用中，这两种模型都是以 JavaScript 对象的形式创建的。Knockout 在视图与数据模型之间建立了一种双向通信机制，从而使得用户输入可以改变应用状态，反过来应用状态又可以影响视图对数据的展示。其中视图模型负责以一种视图（即 HTML）友好的方式来展示数据模型。

HTML 是网页浏览器中的数据展示技术，Knockout 的视图模型既可以直接绑定到已有的 HTML 文档元素，也可以使用由 HTML 模板创建的新元素。Knockout 甚至还能创建可复用的 HTML 组件（拥有各自属性和行为的自定义 HTML 标签）。

本章的 Omnom 菜谱（Omnom Recipes）示例，将菜谱数据（“数据模型”）展示为了一种适合浏览的概要/详细式用户界面。页面的左侧是一个菜谱列表，右侧则呈现详细的菜谱信息，两部分都是非常适合使用 Knockout 视图模型表示的逻辑组件。每一部分都拥有自己的视图模型，程序会自动协调两者之间的交互。最后，用户可能希望可以添加或者编辑菜谱，因此还会介绍其他几种 HTML 标记和视图模型。

清单 7-2 将该示例程序的目录结构展示为了树形（利用 `tree` 命令）。

清单 7-2 示例程序的文件结构

```
example-001$ tree --dirsfirst
```

```
.
├── public
│   ├── scripts
│   │   ├── vendor
│   │   │   ├── jquery-2.1.3.min.js
│   │   │   └── knockout-3.3.0.js
│   │   ├── app.js
│   │   ├── recipe-details.js
│   │   └── recipe-list.js
│   ├── styles
│   │   ├── app.css
│   │   └── index.html
│   ├── index.js
│   └── recipes.json
```

`index.js` 负责启动一个 Web 服务器，以处理对于 `public` 目录中文件的请求。当程序的 Web 页面通过 AJAX 请求菜谱数据时，服务器会序列化 `recipes.json` 中的数据，并将其返回给浏览器。

当用户访问 `http://localhost:8080` 时，`public` 目录中的 `index.html` 将会作为默认的请求文件返回给浏览器。该文件包含许多含有 Knockout 属性的 HTML 标签，还引用了 `public/styles` 中的 `app.css` 样式文件、`public/scripts/vendor` 中的两个第三方脚本，以及 `public/scripts`

下的三个脚本文件。

Knockout 视图模型既可以应用到一个完整的页面上，也可以应用到页面上特定 HTML 元素的范围内。除非是特别简单的应用，否则我们建议使用多个视图模型来保持应用的模块化。在 Omnom 菜谱应用中，程序的 UI 层包含两个逻辑组件：菜谱列表和已选定菜谱的详细视图。这里我们并没有为整个页面创建一个大的视图模型，而是将程序分为两个模块，分别放在 `public/scripts` 目录下的 `recipe-list.js` 和 `recipe-details.js` 中。最终我们在 `app.js` 文件中使用这些视图模型，并协调二者在页面中的活动。

图 7-1 展示的是程序最终运行结果的截屏，左侧是菜谱列表，右侧是菜谱详情。

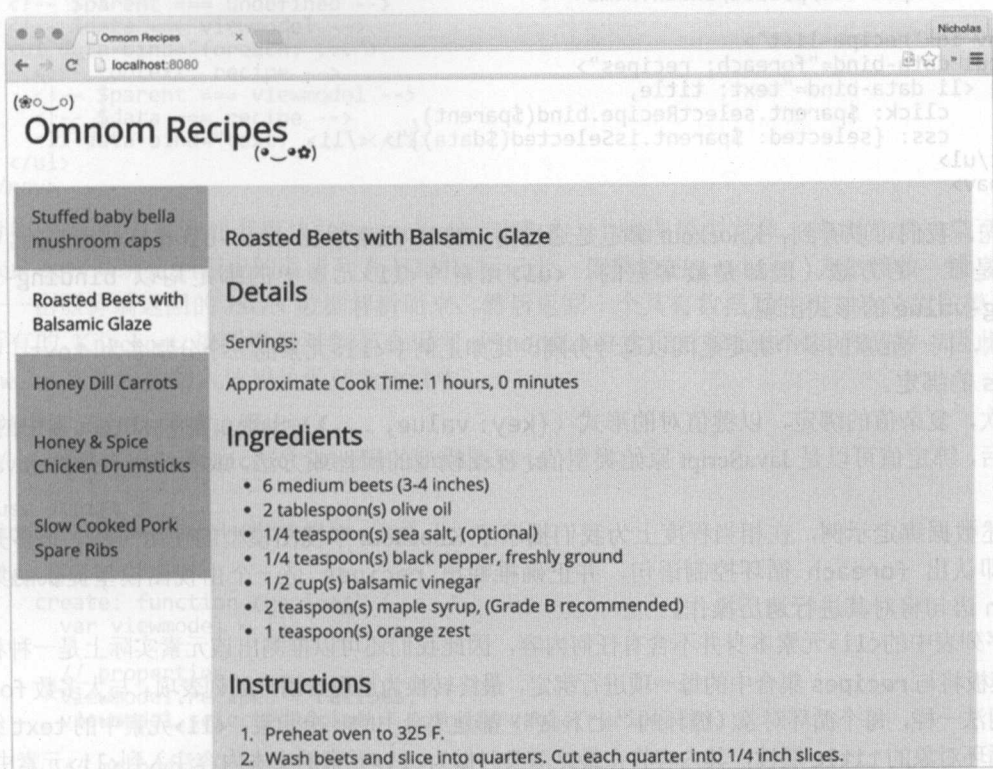


图 7-1 Omnom Recipes 截图

注意 为了避免混淆，该示例应用仅使用了简单的 JavaScript 闭包，而没有使用任何客户端框架或是基于模块的构建工具。这些闭包通常会在全局 `window` 对象上添加一个属性，以便其他脚本调用。例如，`recipe-list.js` 文件会创建一个全局对象 `window.RecipeList` 供 `app.js` 使用。尽管这种方法完全可行，但是我们仍然要了解这样的架构决策是由示例程序的简单的需求所决定的。

7.1.1 菜谱列表

`index.html` 文件包含所有的 HTML 元素以及 Knockout 模板，该文件可以划分为以下三个主要的顶级元素：

- `<header>` 标签，包含不会被 Knockout 处理的静态 HTML 内容

- `<nav id="recipe-list">` 标签, 包含无序的菜谱列表, 由 Knockout 自动生成
- `<section id="recipe-details">` 标签, 显示菜谱的详细信息, 同样由 Knockout 生成

菜谱列表 (`recipe-list`) 元素尽管简单, 但是实际上已经包含了很多不同类型的 Knockout 绑定。下面这段 HTML 代码对应的视图模型将会绑定到 `<nav>` 元素。有鉴于此, 我们通过研究清单 7-3 中的 HTML 标记, 就可以推测出很多 Knockout 的工作细节。

清单 7-3 菜谱列表的标签与绑定语法

```
<!-- example-001/public/index.html -->
```

```
<nav id="recipe-list">
  <ul data-bind="foreach: recipes">
    <li data-bind="text: title,
      click: $parent.selectRecipe.bind($parent),
      css: {selected: $parent.isSelected($data)}"> </li>
  </ul>
</nav>
```

首先, 我们可以看到, Knockout 绑定是通过为 HTML 元素添加 `data-bind` 属性实现的。当然, 这并不是唯一的方法, 但却是最常见的。`` 元素与 `` 元素中的绑定均以 `binding-name: binding-value` 的形式出现。

其次, 同一元素的多个绑定之间以逗号分隔。比如上例中 `` 元素的三个分别名为 `text`、`click` 以及 `css` 的绑定。

再次, 复杂值的绑定, 以键值对的形式 (`{key: value, ...}`) 出现, 例如 `` 元素中的 `css`。

最后, 绑定值可以是 JavaScript 原始类型值、视图模型的属性或方法, 或者任何有效的 JavaScript 表达式。

上述数据绑定示例, 在相当程度上为我们揭示了 Knockout 中视图模型的使用方法。很多开发者能够立即认出 `foreach` 循环控制语句, 并正确推断出 `recipes` 是一个由视图模型提供的集合, `foreach` 语句将对其进行遍历操作。

无序列表中的 `` 元素本身并不含有任何内容, 因此我们还可以推测出该元素实际上是一种模板元素, 该模板将与 `recipes` 集合中的每一项进行绑定, 最终转换为呈现给用户的列表项。与大多数 `foreach` 循环的用法一样, 每个循环对象 (循环的“上下文”) 都是集合中的一个元素。`` 元素中的 `text` 绑定指向当前循环对象的 `title` 属性, 其内容将在最终渲染时做为 `` 元素的文本内容注入到 `` 元素中。

`click` 和 `css` 绑定都引用了特殊的 `$parent` 对象。该对象告诉 Knockout 此位置的绑定上下文并不是当前的循环对象, 而是在 `foreach` 语句中绑定的视图模型。

当用户点击列表触发 `click` 事件时, `click` 绑定将会调用视图模型中的 `selectRecipe()` 方法。注意, 通过向 `bind` 函数传入对特殊变量 `$parent` 的引用, `selectRecipe()` 方法被绑定到了 `$parent` 视图模型 (即 `$recipes`) 上。这将保证在 `selectRecipe()` 方法执行时, 其内部的 `this` 变量指向的不是 `click` 事件处理器被添加处的 DOM 元素。

与此相反, 当 `css` 绑定调用 `$parent` 对象 (视图模型) 上的 `isSelected()` 方法时, 将由 Knockout (而不是 DOM) 管理调用关系, 保证该方法内部的 `this` 变量指向视图模型, 而不是 DOM 元素。

`css` 绑定告诉 Knockout, 当特定条件得到满足时, 为 DOM 元素添加指定的 CSS 类。`css` 绑定的值以“选择器 (selector) / 处理函数”键值对的形式出现, 由 Knockout 在 DOM 元素渲染期间进行求值。如果 `isSelected()` 方法的返回值为 `true`, `selected` 类就会被添加到列表元素 `` 上。接下来我们来看另一个特殊变量 `$data`, 它被作为参数传给了 `isSelected()` 方法。变量 `$data` 总是

指向 Knockout 运行时的当前上下文对象。某些 Knockout 绑定，例如 **text**，默认在当前上下文对象上进行各种操作；另外一些绑定，例如 **foreach**，则会改变程序运行时的上下文对象。

清单 7-4 以 HTML 注释的形式标明了上下文对象以及各个特殊变量的值。

清单 7-4 Knockout 绑定切换上下文

```
<!-- example-001/public/index.html -->

<nav id="recipe-list">
  <!-- context: viewModel -->
  <!-- $parent === undefined -->
  <!-- $data === viewModel -->
  <ul data-bind="foreach: ..."
    <!-- context: recipe -->
    <!-- $parent === viewModel -->
    <!-- $data === recipe -->
    <li data-bind="text: ..."></li>
  </ul>
</nav>
```

清单 7-5 中的菜谱列表模块 (RecipeList) 创建了一个视图模型对象。当浏览器渲染页面时，Knockout 会将其绑定到相应的菜谱列表 HTML 标记上。该模块的 **create()** 方法接收一个菜谱对象数组——由服务端返回的 JSON 数据解析而来，然后返回一个具有数据属性和方法的视图模型对象。几乎所有的 Knockout 视图模型对象都会用到 **window.ko** 全局对象提供的工具函数，因此我们将 **window.ko** 作为参数传入该模块的闭包方法中。

清单 7-5 菜谱列表的视图模型

```
// example-001/public/scripts/recipe-list.js

'use strict';
window.RecipeList = (function (ko) {

  return {
    create: function (recipes) {
      var viewModel = {};

      // properties
      viewModel.recipes = recipes;
      viewModel.selectedRecipe = ko.observable(recipes[0]);

      // methods
      viewModel.selectRecipe = function (recipe) {
        this.selectedRecipe(recipe);
      };

      viewModel.isSelected = function (recipe) {
        return this.selectedRecipe() === recipe;
      };

      return viewModel;
    }
  };
})(window.ko));
```

注意 视图模型对象本身有很多种创建方法。本例中，每个视图模型对象都是一个由工厂函数创建的简单的对象字面量。实践当中，通过 JavaScript 构造函数的方式创建视图模型也很常见。但是视图模型本质上仍然是普通的对象，具体如何创建还要开发者视情况而定。

除了 `selectedRecipe` 属性, 菜谱列表视图模型几无亮点可言: `foreach` 绑定遍历的是 `recipes` 属性 (一个普通的 JavaScript 对象数组); 每个列表项的 `click` 绑定都会调用 `selectRecipe()` 方法 (并传入特定的菜谱对象); 当 Knockout 渲染每个列表项时, 程序通过 `isSelected()` 方法来当前菜谱对象是否等于 `selectedRecipe` 属性。实际上, 这种说法并不完全正确, `selectedRecipe` 并不是一个菜谱对象, 而是一个函数, 一个 Knockout 中的 observable 函数。

Observable 是一种特殊函数, 它可以保存一个值, 并在其发生变化时通知潜在的订阅者。订阅关系在将 HTML 元素绑定到 observable 对象之时自动确立, 并由 Knockout 在后台进行管理。Observable 由全局对象 `ko` 上的特殊工厂函数创建。例如, 在清单 7-5 中, 我们通过调用 `ko.observable(recipes[0])` 创建了一个 observable 对象 `selectedRecipe`。其初始值是 `recipes` 数组的第一个元素。当不传入参数调用 `selectedRecipe()` 时, 程序将返回该 observable 所包含的值 (此处是 `recipes[0]` 对应的对象)。任何传入 `selectedRecipe()` 的值都将覆盖原有的值。尽管 `selectedRecipe()` 并未绑定模板中的元素, 但它却会在用户操作时, 通过调用视图模型中的方法, 影响最终模板的渲染。更新后的值会作为用户输入, 用于展示菜谱详情组件。

7.1.2 菜谱详情

当点击菜谱列表中的某一项时, 菜谱的详情会展示在右侧面板中 (见图 7-1)。清单 7-6 展示了渲染菜谱详情所需的 HTML 元素, 以及相关 Knockout 绑定语句。

清单 7-6 菜谱详情的 HTML 模板

```

<!-- example-001/public/index.html -->
<section id="recipe-details">
  <h1 data-bind="text: title"></h1>

  <h2>Details</h2>
  <p>Servings: <span data-bind="text: servings"></span></p>
  <p>Approximate Cook Time: <span data-bind="text: cookingTime"></span></p>

  <h2>Ingredients</h2>

  <ul data-bind="foreach: ingredients">
    <li data-bind="text: $data"></li>
  </ul>

  <h2>Instructions</h2>

  <ol data-bind="foreach: instructions">
    <li data-bind="text: $data"></li>
  </ol>

  <a data-bind="visible: hasCitation,
    attr: {href: citation, title: title}"
    target="_blank">Source</a>

</section>

```

<h1> 的 `text` 绑定会读取视图模型对象的 `title` 属性, 然后将其渲染到 <h1> 元素内。

因为 “Details” 标题下面几个段落 (`p` 标签) 不仅包含 Knockout 动态生成的内容, 还有很多静态内容, 例如 “Servings:” 和 “Approximate Cook Time:” 等, 所以我们在每个段落中使用 `` 标签来单独维护 Knockout 对于 `servings` 与 `cookingTime` 的绑定。

随后程序通过 `foreach` 遍历了几个字符串数组, 每次遍历的结果可以由 `$data` 获得, 用以生成

列表中每一项的文字内容。

页面底部的标签用于生成菜谱来源的链接。如果菜谱没有来源，则该链接不显示。在本示例中，标签的 `visible` 绑定语句，绑定了视图模型对象的 `hasCitation` 属性；当 `hasCitation` 的值是空时，链接不显示。随后的 `attr` 绑定类似于菜谱列表模板中的 `css` 绑定，接受一个键值对形式的绑定，其中键对应的是元素的属性，值则是视图模型对象对应的属性值。

相比于菜谱列表，菜谱详情的视图模型对象使用了更多的绑定语句。清单 7-7 展示了菜谱列表视图模型对象的大体模式。模块的调用者可以调用 `RecipeDetails.create()` 方法，将选定的菜谱对象作为 `create` 方法的参数传入视图模型对象。该模块同样使用全局对象 `ko` 提供的工具方法，因此我们将其作为参数传入模块的闭包中。

清单 7-7 菜谱详情的视图模型

```
// example-001/public/scripts/recipe-details.js
'use strict';
window.RecipeDetails = (function (ko) {
  return {
    create: function (recipe) {
      var viewModel = {};
      // add properties and methods...
      return viewModel;
    }
  };
})(window.ko));
```

如清单 7-8 所示，针对菜谱对象的每个属性，菜谱详细的视图模型都提供了一个与之对应的 `observable` 属性。只有在关注视图模型的属性变化时，才需要设置为 `observable` 属性。对于那些静态的、不会变化的属性，我们可以直接使用简单的 JavaScript 属性和值。我们这里使用 `observable`，是因为在菜谱详情的视图模型中，只有一个视图模型的实例。当我们在菜谱列表中选择一个菜谱时，菜谱对象会更新菜谱详情的视图模型，随后根据设置的 `observable`，对应的 HTML 元素也会被立即更新。

清单 7-8 菜谱列表视图模型用到的属性

```
// example-001/public/scripts/recipe-details.js
// properties
viewModel.title = ko.observable(recipe.title);
viewModel.servings = ko.observable(recipe.servings);
viewModel.hours = ko.observable(recipe.cookingTime.hours);
viewModel.minutes = ko.observable(recipe.cookingTime.minutes);
viewModel.ingredients = ko.observableArray(recipe.ingredients);
viewModel.instructions = ko.observableArray(recipe.instructions);
viewModel.citation = ko.observable(recipe.citation);

viewModel.cookingTime = ko.computed(function () {
  return '$1 hours, $2 minutes'
    .replace('$1', this.hours())
    .replace('$2', this.minutes());
}, viewModel);
```

清单 7-8 展示了两种新的 `observable`：`ko.observableArray()` 与 `ko.computed()`。

`observable` 数组可以监听数组中元素的增加、移除以及位置更改等变化，这样在数组发生变化时，其订阅者将会收到通知。虽然我们的例子中并没有涉及数组的变化，但我们会在后续章节中介绍如何实现集合的绑定。

可计算的 `observable` 会根据视图模型中已有的 `observable` 计算出一个值。`ko.computed()` 方法的第一个参数是一个回调方法，用于计算该 `observable` 的值；第二个是可选参数，可定义回调函数被调

用时的 `this` 指向的上下文对象。在清单 7-8 中，我们以格式化字符串的形式连接视图模型中的 `hours` 与 `minutes` 属性，最终得出 `cookingTime` 属性。当 `hours` 或者 `minutes` 属性的值发生变化时，`cookingTime` 的值也会随之发生变化。

■ **注意** 尽管在 Knockout 绑定语句中，`hours` 与 `minutes` 被作为属性使用，但两者实际上都是函数。因此在可计算 observable 中，我们必须通过函数调用方式来获取它们的值。

在下面的清单 7-9 中，菜谱详情视图模型中的方法都比较简单。`hasCitation()` 方法检验 `citation` 属性的值是否为空，`update()` 方法则接收一个菜谱对象，并用该对象更新现有的 observable 属性。该方法虽然没有绑定到视图，但是当列表视图有新的菜谱被选中时将会被调用。

清单 7-9 菜谱详情视图模型中用到的方法

```
// example-001/public/scripts/recipe-details.js
// methods
viewmodel.hasCitation = function () {
    return this.citation() !== '';
};

viewmodel.update = function (recipe) {
    this.title(recipe.title);
    this.servings(recipe.servings);
    this.hours(recipe.cookingTime.hours);
    this.minutes(recipe.cookingTime.minutes);
    this.ingredients(recipe.ingredients);
    this.instructions(recipe.instructions);
    this.citation(recipe.citation);
};
```

7.2 将视图绑定到 DOM

前文提到的两个工厂方法，都在全局 `window` 对象上创建了视图模型的引用。而在清单 7-10 中，我们展示的是在 `app.js` 文件中如何使用两个视图模型，并将它们展示在页面中。

清单 7-10 绑定视图模型到 DOM

```
// example-001/public/scripts/app.js
(function app ($, ko, RecipeList, RecipeDetails) {
    // #1
    var getRecipes = $.get('/recipes');

    // #2
    $(function () {
        // #3
        getRecipes.then(function (recipes) {
            // #4
            var list = RecipeList.create(recipes);
            // #5
            var details = RecipeDetails.create(list.selectedRecipe());
            // #6
            list.selectedRecipe.subscribe(function (recipe) {
                details.update(recipe);
            });
            // #7
            ko.applyBindings(list, document.querySelector('#recipe-list'));
            ko.applyBindings(details, document.querySelector('#recipe-details'));
        });
    });
});
```



```

    }).fail(function () {
        alert('No recipes for you!');
    });
});
})(window.jQuery, window.ko, window.RecipeList, window.RecipeDetails));

```

app 模块用来从服务器加载和初始化菜谱数据，等待 DOM 进入就绪状态，然后创建视图模型的实例，并绑定到对应的 HTML 元素之上。下面介绍清单 7-10 中注释的每一步内容（对应代码中的 `// #1` 等注释）。

1. 创建一个 jQuery 的 Ajax Promise，等待 GET `/recipes` 请求获取的数据返回。
2. 传入 `$()` 的方法会在整个 DOM 树就绪之后执行，从而保证在 Knockout 执行绑定语句之前，所有的模板都已就绪。
3. 当步骤 1 中创建的 promise 决议之后，将返回的数据将传递给对应的决议处理函数（resolution handler）。如果 promise 失败，则通过 `alert` 告诉用户请求的过程出了问题。
4. 成功加载的数据将会传入 `RecipeList.create()` 方法，而该方法的返回值就是我们菜谱列表的视图模型。
5. 创建菜谱详情视图模型的过程是类似的，工厂方法接收一个菜谱对象作为参数，而这里传入的是菜谱列表视图模型的 `selectedRecipe` 属性（默认情况下，`selectedRecipe` 的值是菜谱列表中的第一个菜谱对象）。
6. 创建菜谱详情视图模型之后，我们将设置它监听菜谱列表 `selectedRecipe` 属性的变化。当该属性发生变化时，会调用传入 `subscribe()` 的回调方法，而 `selectedRecipe` 的新值将被作为参数传入回调方法。在回调方法中，我们将新选择的菜谱对象传入菜谱详情的 `update()` 方法，随即更新菜谱详情的展示。
7. 最后，我们通过全局的 `ko.applyBindings()` 方法将视图模型绑定到 DOM 元素上。在清单 7-10 中，我们可以看到 `applyBindings()` 方法接收两个参数：绑定的视图模型与被绑定的 DOM 元素。调用这个方法之后，Knockout 将遍历这个 DOM 元素及其子元素，并一一进行绑定。如果没有指定视图模型绑定的 DOM 元素，那么这个视图模型将会作用到整个页面。这种方法对于简单页面也许有效，但是要应对更复杂的场景，使用多视图模型将会是更好的选择。

7.3 视图模型与表单

Knockout 的视图模型同样可以作用到表单组件中。很多控件如 `<input>` 元素，可以使用标准的 `value` 属性进行绑定，同时也有一些针对 `<select>` 之类的特殊元素的绑定。例如，我们可以在 `<select>` 标签中设置 `options` 绑定，以实现动态创建 `<option>` 标签。总体来说，表单域的绑定很像示例代码里的，但复杂的表单下，可能会变得非常棘手，还需要更多富有创造性的绑定策略。

这一部分的例子将对前面提到的菜谱详情视图模型加以扩展，加入一个“编辑”模式。这样用户在预览某个菜谱的详情时，就可以通过表单对现有菜谱对象进行修改。我们仍然使用现有的视图模型，但会加入很多表单元素，复杂度也会随之提升。

7.3.1 切换到“编辑”模式

我们在菜谱详情的页面中加入了三个按钮。图 7-2 与图 7-3 演示了按钮将会如何显示在页面上的样子。

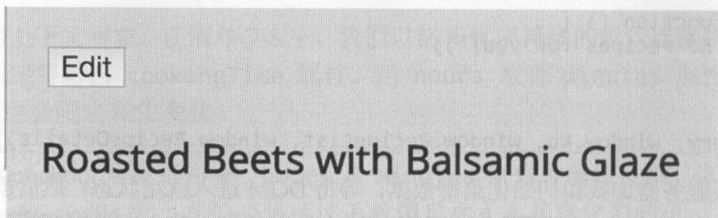


图 7-2 在“预览”模式下，我们可以看到 Edit 按钮

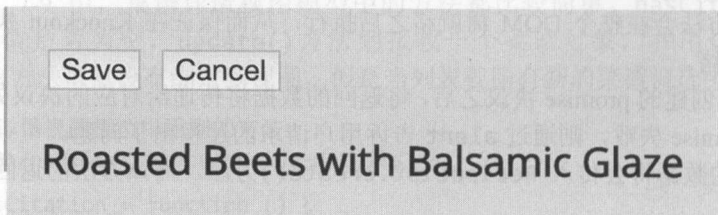


图 7-3 在“编辑”模式下，我们可以看到 Save 按钮与 Cancel 按钮

单击 Edit 按钮，应用将会由“预览”模式切换为“编辑”模式，并以表单的形式展示菜谱的每个属性以供编辑。同时 Edit 按钮会隐藏，而 Save 按钮与 Cancel 按钮将变得可见。此时，如果用户单击 Save 按钮，则对菜谱的更改将会被保存。如果用户单击了 Cancel 按钮，所有更改都会被放弃，菜谱详情也会还原至初始状态。

清单 7-11 展示了代码中如何控制几个按钮的显示与隐藏。

清单 7-11 编辑按钮的模板

```
<!-- example-002/public/index.html -->
<div>
  <!-- in read-only view -->
  <button data-bind="click: edit, visible: !isEditing()">Edit</button>
  <!-- in edit view -->
  <button data-bind="click: save, visible: isEditing">Save</button>
  <button data-bind="click: cancelEdit, visible: isEditing">Cancel</button>
</div>
```

首先，我们为每个<button>添加了 click 绑定：edit()、save()、cancelEdit()。与之前的案例不同，这些方法并没有使用 bind() 来指定视图模型中的 this，而是如清单 7-12 所示，直接在方法内部使用了 viewmodel 对象。我们在下面的例子中展示了新增的属性与方法，出于简洁的考虑，省略了 recip-list.js 文件中没有改变的部分。

清单 7-12 使用 viewmodel 对象的方法，而非 this

```
// example-002/public/scripts/recipe-details.js
// properties
viewModel.previousState = null;
viewModel.isEditing = ko.observable(false);

// methods
viewModel.edit = function () {
  viewModel.previousState = ko.mapping.toJS(viewModel);
  viewModel.isEditing(true);
};
viewModel.save = function () {
```

```

// TODO save recipe
viewmodel.isEditing(false);
};

viewmodel.cancelEdit = function () {
    viewmodel.isEditing(false);
    ko.mapping.fromJS(viewmodel.previousState, {}, viewmodel);
};

```

由于通过闭包的形式创建视图模型，它内部的方法通常都可以通过名字调用。通过避免使用 `this`，事件绑定得以简化，也避免了一些潜在的 bug。

另外，每个 `<button>` 都有一个 `visible` 属性，绑定视图模型的 `isEditing` 属性，但是只有 `Edit` 按钮直接以方法的形式获取 `isEditing` 的值。同时，也只有这里用到了取反操作符 (!)，目的是将绑定语句转换为表达式。在表达式中，所有 `observable` 都必须所作为函数来执行，以获取其值。如果数据监测的是它本身的值，如本例中 `Save` 与 `Cancel` 的 `visible` 绑定语句，则会由 `Knockout` 统一维护，自动获取正确的值。

`edit()`、`save()` 和 `cancelEdit()` 三个方法都会对 `isEditing` 的值进行操作，以控制表单中不同按钮的显示（以及我们将要看到的，控制不同表单组件的显示）。整个编辑过程始于 `Edit()` 方法的调用，终于用户执行保存或者放弃操作。

为了确保用户放弃编辑之后，界面可以正确地恢复到原来的状态。在 `edit()` 方法中，我们将视图模型序列化之后保存在 `previousState` 属性中。这样，当用户放弃编辑操作时，我们可以反序列化出之前的状态，恢复每一个属性。

这里，我们用到了 `Knockout` 的 `mapping` 插件来完成序列化、反序列化的过程：

```

// 序列化 视图模型
viewmodel.previousState = ko.mapping.toJS(viewmodel);
// 反序列化 视图模型
ko.mapping.fromJS(viewmodel.previousState, {}, viewmodel);

```

■ **提示** `Knockout` 的 `mapping` 插件是独立于 `Knockout` 的核心库的。最新的版本可以到 <http://knockoutjs.com/documentation/plugins-mapping.html> 下载。要使用这个插件，只需在引用 `Knockout` 的 `<script>` 标签之后添加一个指向 `mapping` 插件脚本的 `<script>` 标签即可。它会自动创建 `ko.mapping` 属性，并将其挂载到全局变量 `ko`。

`mapping` 插件可以序列化/反序列化包含 `observable` 属性的对象，在序列化时自动读取属性的值，在反序列化时则会自动创建属性的值。当我们在 `edit()` 方法中调用 `ko.mapping.toJS(viewmodel)` 时，它会创建一个简单 JavaScript 对象，对象的属性名对应视图模型的属性名，但是其值不再是 `observable` 函数，而是简单的 JavaScript 的值。当我们放弃修改，将这个对象再放回视图模型时，我们需要调用 `ko.mapping.fromJS()` 方法。该方法有三个参数：

- 将要写入视图模型的一个简单 JavaScript 对象
- 一个对象，表示的是第一个参数中 JavaScript 对象的属性与视图模型中 `observable` 属性的映射关系。如果传入一个空对象，则表示两个对象属性的名字一一对应
- 接受对象字面量的视图模型

■ **注意** `Knockout` 的 `mapping` 插件，通过调用其 `toJS()` 和 `fromJS()` 函数，将视图模型序列化/反序列化成简单 JavaScript 对象字面量，同时也可以通过 `toJSON()` 方法与 `fromJSON()` 方法进行 JSON 字符串的转换。这些方法在应对 JSON 格式数据 CRUD（增 create，删 delete，改 update，查 query）操作的场合下非常有用。

尽管我们看到了 Save 按钮，但是真正的操作逻辑将在后文的示例中给出。

7.3.2 更改菜谱的标题

事实上，不管菜谱详情是在编辑模式还是只读模式，菜谱标题始终会显示，只是当用户单击 Edit 按钮时，标题下方会出现一个<label>标签以及一个<input>输入区域。在这两个标签的外层有一个<div>，通过 visible 属性与视图模型的 isEditing 进行关联，以控制这两个标签的展示与否。<input>输入框通过 value 绑定，将其 value 与视图模型的 title 属性进行绑定。默认情况下，<input>的值只有在失去焦点时才会触发 observable 的变化，因此在清单 7-13 中，由于标题与<input>共同绑定在视图模型中的 title 属性上，当输入控件失去焦点，标题内容会立即更新为<input>的值，最终的效果如图 7-4 所示。

清单 7-13 更改菜谱标题

```
<!-- example-002/public/index.html -->
<h1 data-bind="text: title"></h1>
<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">
  <label for="recipe-title">Title:</label>
  <input data-bind="value: title" name="title" id="recipe-title" type="text" />
</div>
```

Roasted Beets with Balsamic Glaze

Title:

Roasted Beets with Balsamic Glaze

图 7-4 编辑菜单标题

7.3.3 更改菜谱的份量和烹饪时间

我们可以看到，在清单 7-14 中，菜谱的只读项 Serving 在进入编辑模式后会被隐藏。取而代之的是一个<select>元素，供用户调整烹饪的菜量。同理，通过监测 isEditing 属性来判断元素的显示和隐藏。

清单 7-14 烹饪数量的修改

```
<!-- example-002/public/index.html -->
<h2>Details</h2>
<!-- in read-only view -->
<p data-bind="visible: !isEditing()">
  Servings: <span data-bind="text: servings"></span>
</p>
<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">
  <label for="recipe-servings">Servings:</label>
  <select data-bind="options: servingSizes,
    optionsText: 'text',
    optionsValue: 'numeral',
    value: servings,
    optionsCaption: 'Choose...'"
    name="recipeServings"
    id="recipe-servings">
  </select>
</div>
```


清单 7-14 中，我们在`<select>`中用了一些新的 Knockout 元素绑定方法来控制视图模型数据。`options` 绑定告诉 Knockout，`<select>`可以通过哪个属性获取创建`<option>`的数据集。在本示例中使用 `servingSizes`，其值是一个简单 JavaScript 数组。

对于 `options` 绑定来说，数组的每一项如果是基本数据类型如 `number`、`string`，那么 Knockout 会指定这一项同时作为`<option>`元素的 `text` 与 `value`。而如果每一项是一个对象，那么我们需要通过 `optionsText` 与 `optionsValue` 来指定通过哪个属性来获取 `text` 与 `value` 的值。在清单 7-15 中，`text` 属性是数字的文字表示，而 `numeral` 则是具体的数字值。当用户选择了某个份量之后，对应的数字值就会被传入 `viewmodel.servings()`。

清单 7-15 视图模型中的菜谱配比数据

```
// example-002/public/scripts/recipe-details.js
// properties
viewmodel.servings = ko.observable(recipe.servings);
viewmodel.servingSizes = [
  {text: 'one', numeral: 1},
  {text: 'two', numeral: 2},
  {text: 'three', numeral: 3},
  {text: 'four', numeral: 4},
  {text: 'five', numeral: 5},
  {text: 'six', numeral: 6},
  {text: 'seven', numeral: 7},
  {text: 'eight', numeral: 8},
  {text: 'nine', numeral: 9},
  {text: 'ten', numeral: 10}
];
```

`<select>` 标签将下拉菜单选择的值与视图模型的值绑定在一起。当`<select>`被渲染时，该值在 DOM 中自动被选择并呈现给用户；当用户选择新的份量时，视图模型绑定的数值也将被更新。

最后，`optionsCaption` 绑定会在 DOM 中创建特殊的`<option>`元素，作为第一项展现在下拉菜单中。不过不用担心，它永远都不会被设置为视图模型的选中值。它仅仅是友好的展现，提示用户下拉菜单如何使用。

图 7-5 和图 7-6 演示了折叠和展开模式下的下拉菜单。

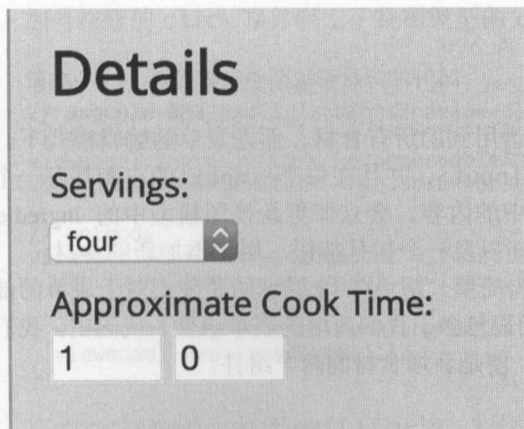


图 7-5 含有初始值的下拉菜单

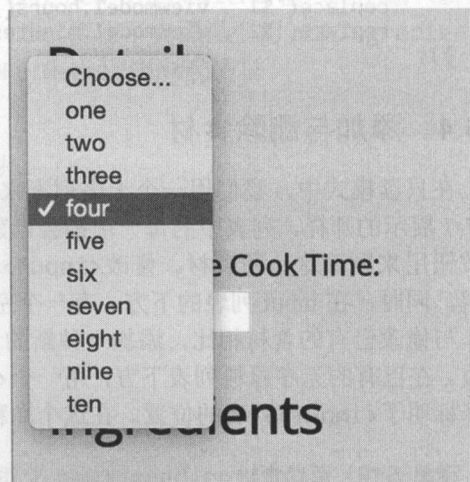


图 7-6 在份量 (Serving) 下拉框中选择一个新值

在图 7-5 中，我们还看到了烹饪时间（cooking time）区域。这里没有用到任何特殊的绑定，只是简单的输入框，通过简单的 value 绑定来维护视图模型。它们的显示与隐藏用到了与前文提到的完全一样的方法。

清单 7-16 Cooking Time 模板

```
<!-- example-002/public/index.html -->
<!-- in read-only view -->
<p data-bind="visible: !isEditing()">
  Approximate Cook Time: <span data-bind="text: cookingTime"></span>
</p>
<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">
  <label for="recipe-hours">Approximate Cook Time:</label>
  <input data-bind="value: hours"
    name="hours"
    id="recipe-hours"
    type="number" />
  <input data-bind="value: minutes"
    name="minutes"
    id="recipe-minutes"
    type="number" />
</div>
```

回忆一下，前文我们提到 cookingTime 是由 hours 与 minutes 通过可计算 observable 的形式计算出来的。在清单 7-17 中，我们用到了同样的方法，当输入框中的内容发生改变时，最终的结果也会随之发生改变。同时也应该注意到，在 computed observable 中，我们没有再使用 this，而是使用具体的 viewmodel 对象。

清单 7-17 视图模型中的 Hours、Minutes 与以及计算得到的 Cooking Time

```
// example-002/public/scripts/recipe-details.js
// properties
viewmodel.hours = ko.observable(recipe.cookingTime.hours);
viewmodel.minutes = ko.observable(recipe.cookingTime.minutes);
viewmodel.cookingTime = ko.computed(function () {
  return '$1 hours, $2 minutes'
    .replace('$1', viewmodel.hours())
    .replace('$2', viewmodel.minutes());
});
```

7.3.4 添加与删除食材

在只读模式中，我们用一个无序列表来展示菜谱用到的所有食材。而在表单的编辑模式下，如图 7-7 展示的那样，列表中的每一项都会创建一个，并且在每个的右侧都有一个减号按钮用来删除这一种食材。修改输入框中的内容，会立即更新视图模型中的 ingredients 数组。同时，在 input 列表的下方，有一个空的 input 以及一个加号按钮，用于添加新的食材。

与修改已有的食材相比，添加一种新的食材更为简单。清单 7-18 展示的是我们对于表单的部分修改。在已有的无序原料列表下方，用一个<div>元素包含了我们新增的表单组件。代码中，我们用注释标明了列表的位置。在这个注释下面，便是新增食材的两个组件。

清单 7-18 新建食材

```
<!-- example-002/public/index.html -->
<h2>Ingredients</h2>
```

```

<!-- in read-only view -->
<ul data-bind="foreach: ingredients, visible: !isEditing()">
  <li data-bind="text: $data"></li>
</ul>
<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">

  <!-- ingredient list inputs here... -->

  <input data-bind="value: newIngredient"
        type="text"
        name="new-ingredient"
        id="recipe-new-ingredient"/>
  <button data-bind="click: commitNewIngredient"
        class="fa fa-plus"></button>
</div>

```

Ingredients

6 medium beets (3-4 inches)

2 tablespoon(s) olive oil

1/4 teaspoon(s) sea salt, (optional)

1/4 teaspoon(s) black pepper, freshly ground

1/2 cup(s) balsamic vinegar

2 teaspoon(s) maple syrup, (Grade B recommended)

1 teaspoon(s) orange zest

+

图 7-7 创建与编辑食材列表

清单 7-19 展示的是我们对视图模型的修改。用户可以在 `new-ingredient` 输入框中输入新的菜谱食材，并单击加号按钮以添加新的食材。`<input>` 的 `value` 绑定了视图模型的 `newIngredient` 属性，加号按钮的 `click` 事件绑定了视图模型的 `commitNewIngredient()` 方法。

清单 7-19 视图模型中添加新原料的代码

```

// example-002/public/scripts/recipe-details.js
// properties
viewModel.ingredients = ko.observableArray(recipe.ingredients);
viewModel.newIngredient = ko.observable('');

// methods
viewModel.commitNewIngredient = function () {
  var ingredient = viewModel.newIngredient();
  if (ingredient === '') return;
  viewModel.ingredients.push(ingredient);
  viewModel.newIngredient('');
};

```

在 `commitNewIngredient()` 方法中，程序首先获取了 `newIngredient` 的值，并判断值是否为空。如果值为空，也就是用户没有输入内容，程序则直接返回，否则会将 `newIngredient` 的值加入

`ingredients` 数组并清空 `newIngredient` 的内容。

■ **提示** Knockout 的 observable 数组扩展了原生 JavaScript 数组的方法，如 `push()`、`pop()`、`slice()`、`splice()` 等。当调用这些重写之后的方法时，observable 数组的订阅者将会收到数组更新的通知。

当我们在 `ingredients` 数组中添加一种新的食材时，Knockout 会自动维护 DOM 的变化。在编辑模式下，原料列表虽然不可见，但是依然会有一个新的列表项默默地加入食材列表。同时在 `<input>` 列表中，我们也会看到新增了一项。

清单 7-20 食材列表

```
<!-- example-002/public/index.html -->
<h2>Ingredients</h2>
<!-- in read-only view -->
<ul data-bind="foreach: ingredients, visible: !isEditing()">
  <li data-bind="text: $data"></li>
</ul>
<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">
  <ul data-bind="foreach: ingredients" class="listless">
    <li>
      <input data-bind="value: $data,
        valueUpdate: 'input',
        attr: {name: 'ingredient-' + $index()},
        event: {input: $parent.changeIngredient.bind($parent, $index())}"
        type="text" />
      <button data-bind="click: $parent.removeIngredient.bind($parent, $index())"
        class="fa fa-minus"></button>
    </li>
  </ul>

  <!-- new ingredient input here... -->
</div>
```

Knockout 遍历 `ingredients` 数组的每一项，创建了一个 `<input>` 无序列表。在 `foreach` 循环中，我们可以用 `$data` 获取每一项的值，从而将 `<input>` 的 `value` 与 `$data` 绑定到一起。同时我们可以用 `$index` 获取循环的索引，因此我们通过 `attr` 绑定语句，连接 `ingredient-` 与 `$index`，为每一个 `<input>` 创建 `name` 属性。如前所述，`attr` 中用到了绑定表达式，因此需要用方法的形式获取 `$index` 的值。

需要特别注意的是，observable 数组只作用到数组本身，并不会关心数组中的元素。当每一个食材对象通过 `$data` 绑定到 `<input>` 时，它并不会通知父层级的 observable 数组；当 `$data` 中的数据发生变化时，数组并不会察觉，数组中维护的数据也不会变化。这确实不够友好，但也还是有一些应对策略。

第一种方案，我们可以为 observable 数组的每个元素添加一个 observable 属性（如 `{ ingredient: ko.observable('20 mushrooms') }`），并将每个 `<input>` 的 `value` 绑定到对象的 `$data.ingredient` 属性。尽管 observable 数组依然不知道包含的数据的变化，但因为每个元素都通过 observable 的方式建立了数据绑定，自然也就可以维护这个变化。

清单 7-20 中提到的是第二种方案，我们为 `<input>` 添加了对 `input` 事件的绑定。当 `<input>` 中输入的数据发生变化时，`input` 事件就会触发，与之关联的 `changeIngredient()` 方法也就会被调用，在方法中完成了对 `ingredients` 数组的修改。

这两种方法并没有好坏，都是各有优缺点。

默认情况下，只有在元素失去焦点时，\$data 才会发生变化。而根据例子中 valueUpdate 绑定语句的设置，当元素的 input 事件触发时，Knockout 就会去更改\$data。默认情况下，Knockout 会将当前项的\$data 传入 changeIngredient()方法，但因为这个数据已经覆盖了旧数据，我们还需要获取\$data 在数组中的索引，所以这里用 bind 语句将\$index 作为参数传入 changeIngredient()方法，最终 changeIngredient()方法的第一个参数会是\$index，第二个参数会是\$data。

清单 7-21 中的代码演示了我们如何通过调整成员索引值，更改 ingredients 数组中的数据。

清单 7-21 在视图模型中修改原料

```
// example-002/public/scripts/recipe-details.js
// properties
viewModel.ingredients = ko.observableArray(recipe.ingredients);

// methods
viewModel.changeIngredient = function (index, newValue) {
    viewModel.ingredients()[index] = newValue;
};
```

不幸的是，当一个 observable 数组的原数组模式发生变化时，这个 observable 数组并不知道，其订阅者也就得不到数据更新的通知，也就意味着，DOM 元素不会有任何的变化。在清单 7-22 中，我们提供了一种解决方案，在视图模型中监听 observable 属性 isEditing 的变化。当 isEditing 的值变为 false 时，就意味着用户保存了所有的更改，或者放弃了所有的修改。所以在这里调用 observable 数组 ingredients 的 valueHasMutated()方法，ingredients 数组的订阅者也就收到数据变化的通知，这样就保证在预览模式下，食材列表始终会被精确地更新。

清单 7-22 原数组发生变化时，强制更新 Observable 数组

```
// example-002/public/scripts/recipe-details.js
// properties
viewModel.isEditing = ko.observable(false);
viewModel.isEditing.subscribe(function (isEditing) {
    if (isEditing) return;
    // force refresh
    //
    viewModel.ingredients.valueHasMutated();
});
```

接下来，在菜谱原料数组的每一行旁边是一个减号按钮，按钮的 click 事件绑定 removeIngredient()方法。这个方法类似于 changeIngredient()方法，接收\$index 参数用于识别哪一个元素将要被移除。在清单 7-23 中，可以看到，我们调用了 observable 数组的 splice()方法，根据元素的索引来删除一种菜谱原料。我们用这个方法替代原生数组的 splice()方法，确保数组发生变化时，DOM 元素会立即做出反应。

清单 7-23 删除一种食材

```
// example-002/public/scripts/recipe-details.js
// properties
viewModel.ingredients = ko.observableArray(recipe.ingredients);

// methods
viewModel.removeIngredient = function (index) {
    viewModel.ingredients.splice(index, 1);
};
```

7.3.5 操作步骤

菜谱的操作步骤与食材列表非常相似，但同时也有两个显著的不同之处。首先，因为菜谱的操作步骤是有顺序的，所以步骤列表是一个有序列表。其次，步骤的每一项是可以调整其在列表中的位置的。图 7-8 展示的是步骤内容的输入框，以及相关的操作按钮。

图 7-8 创建与编辑操作步骤

在清单 7-24 中，我们可以看到，步骤列表的实现方案与食材列表极为类似。因此，对于相似的部分，如添加步骤、删除步骤与修改已有步骤，我们不再过多讨论。可以看到，步骤列表中某一步骤位置的上升与下降，是通过列表中新加的向上与向下按钮实现的。

清单 7-24 步骤列表模板

```
<!-- example-002/public/index.html -->
<h2>Instructions</h2>
<!-- in read-only view -->
<ol data-bind="foreach: instructions, visible: !isEditing()">
  <li data-bind="text: $data"></li>
</ol>

<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">
  <!-- existing instructions -->
  <ul data-bind="foreach: instructions" class="listless">
    <li>
      <input data-bind="value: $data,
        valueUpdate: 'input',
        attr: {name: 'instruction-' + $index()},
        event: {input: $parent.changeInstruction.bind($parent, $index())}"
        type="text" />
      <button data-bind="click: $parent.demoteInstruction.bind($parent, $index())"
        class="fa fa-caret-down"></button>
      <button data-bind="click: $parent.promoteInstruction.bind($parent, $index())"
        class="fa fa-caret-up"></button>
      <button data-bind="click: $parent.removeInstruction.bind($parent, $index())"
        class="fa fa-minus"></button>
    </li>
  </ul>

  <!-- new instruction input here... -->
</div>
```

与减号按钮类似，向上按钮与向下按钮都通过 `click` 绑定了视图模型中的方法，并将关联元素的索引作为参数，传入方法。

清单 7-25 展示的是这两个方法的具体实现。`promoteInstruction()` 方法包含以下步骤：首先判断 `index` 是否是 0，如果是 0，也就是说正处于第一个步骤，方法会立即返回，因为第一个步骤无需提升顺序。随后根据传入的 `index`，调用 `instructions` 数组的 `splice()` 方法，移除当前步骤，并获取这个被移除的步骤。随后计算新的 `index`（假设 `index` 从 1 升为 2），继续调用 `splice()` 方法将刚才移除的步骤插入 `instructions` 数组的新位置。`demoteInstruction()` 方法则刚好相反，它首先阻止位于最后一位的步骤继续下降，然后通过两次 `splice()` 操作将其位置向下移动一位。因为用到了 `observable` 数组 `instructions` 提供的方法，因此这两个操作都会立即体现到 DOM 元素的变化。

清单 7-25 在视图模型中提升或者下降步骤的顺序

```
// example-002/public/scripts/recipe-details.js
// properties
viewModel.instructions = ko.observableArray(recipe.instructions);

viewModel.promoteInstruction = function (index) {
  if (index === 0) return;
  var instruction = viewModel.instructions.splice(index, 1);
  var newIndex = index - 1;
  viewModel.instructions.splice(newIndex, 0, instruction);
};

viewModel.demoteInstruction = function (index) {
  var lastIndex = (viewModel.instructions.length - 1);
  if (index === lastIndex) return;
  var instruction = viewModel.instructions.splice(index, 1);
  var newIndex = index + 1;
  viewModel.instructions.splice(newIndex, 0, instruction);
};
```

7.3.6 引文

相比较前面复杂的食材列表与操作步骤列表，引文区域可以说是比较普通的。这里通过 `<input>` 输入框的 `value` 绑定语句关联了视图模型中的 `citation` 属性。最终的效果如图 7-9 所示。

Citation:

<http://www.paleoplan.com/2011/06-09/roasted-beets-with-balsa>

图 7-9 更新菜谱引文

在清单 7-26 中，我们可以看到，代码中通过 `visible` 绑定控制引文链接的显示，而 `visible` 的值是一个复杂的表达式，最终实现的效果是当页面处于预览模式，并且菜谱详情包含引文链接的数据，引文链接才会显示。

清单 7-26 菜谱引文模板

```
<!-- example-002/public/index.html -->
<a data-bind="visible: hasCitation() && !isEditing(),
  attr: {href: citation, title: title}"
  target=" _blank">Source</a>
<div data-bind="visible: isEditing" class="edit-field">
```

```
<label>Citation:</label>
<input name="citation" type="text" data-bind="value: citation" />
</div>
```

7.4 自定义组件

Knockout 仿照著名的 webcomponents.js(<http://webcomponents.org>), 提供了一套完整的自定义组件系统, 以实现代码的重用。

在前面的例子中, 我们在菜谱详情中创建了两个可编辑的列表: 食材列表与操作步骤列表。这两个列表的 HTML 模板与视图模型属性有很多相似的地方, 因此我们就可以将两者抽象为一个组件, 以替换程序中的列表。组件的使用方法请看清单 7-27。

清单 7-27 使用 input-list 组件

```
<!-- example-003/public/index.html -->
<!-- editable ingredients list -->
<input-list params="items: ingredients,
               isOrdered: false"></input-list>

<!-- ... -->

<!-- editable instructions list -->
<input-list params="items: instructions,
                   isOrdered: true"></input-list>
```

一个 Knockout 组件需要包含以下几个部分:

- 一个工厂方法, 用于在页面中创建组件的实例
- HTML 模板, 用于定义组件使用的模板及其包含的绑定关系
- 自定义组件如何注册, 这样 Knockout 在页面中发现自定义组件时, 可以知道如何去实例化这个组件

7.4.1 input-list 组件的视图模型

菜谱详情的视图模型包含了操作食材列表与步骤列表的所有代码, 我们可以将这些代码移到一个公共模块 `input-list.js`, 也就是我们的 `input-list` 组件。

清单 7-28 展示的是 `input-list` 模块的部分代码。这个模块采用了与其他模块类似的代码模式, 在全局 `InputList` 对象上暴露了一个 `create()` 方法。这个工厂方法接收一个参数 `params`。这个参数是一个对象, 用于配置我们的 `input-list` 组件。`params.items` 将作为组件中 `observable` 数组的原始数组, 而 `params.isOrdered`, `params.enableAdd`, `params.enableUpdate`, 以及 `params.enableRemove` 等其他参数则决定了组件渲染时的一些细节。

`defaultTo()` 作为一个辅助函数, 在对象的某个属性取不到值时, 将返回一个默认值。

清单 7-28 input-list 组件的视图模型

```
// example-003/public/scripts/input-list.js
'use strict';
window.InputList = (function (ko) {

    function defaultTo(object, property, defaultValue) { /*...*/ }
```



```

return {
  create: function (params) {
    var viewmodel = {};

    // properties
    viewmodel.items = params.items; // the collection
    viewmodel.newItem = ko.observable('');

    viewmodel.isOrdered = defaultTo(params, 'isOrdered', false);
    viewmodel.enableAdd = defaultTo(params, 'enableAdd', true);
    viewmodel.enableUpdate = defaultTo(params, 'enableUpdate', true);
    viewmodel.enableRemove = defaultTo(params, 'enableRemove', true);

    // methods
    viewmodel.commitNewItem = function () { /*...*/ };
    viewmodel.changeItem = function (index, newValue) { /*...*/ };
    viewmodel.removeItem = function (index) { /*...*/ };
    viewmodel.promoteItem = function (index) { /*...*/ };
    viewmodel.demoteItem = function (index) { /*...*/ };

    return viewmodel;
  }
};

}(window.ko));

```

在清单 7-27 中，我们用到了组件的 `params.items` 与 `params.isOrdered` 属性。当我们在页面中使用一个自定义组件时，绑定的值借助 `params` 对象，传入组件的视图模型中。在我们的例子中，`input-list` 组件获取了菜谱详情视图模型中的 `ingredients` 数组与 `instructions` 数组。

在清单 7-28 中，我们可以看到 `input-list` 组件的方法与清单 7-25 中的极为相似，这里我们没有用到 `ingredients` 数组或者 `instructions` 数组，而是用一个抽象的数组 `items` 来替代它们。`newItem` 属性则绑定了新建项目的 `value`，作用类似于 `recipe-details.js` 文件中的 `newIngredient` 属性与 `newInstruction` 属性。

因为我们将要在页面中使用 `input-list` 组件来处理食材列表与操作步骤列表，所以我们可以移除菜谱详情中与之关联的属性及方法。

7.4.2 input-list 模板

一个可重用的组件通常包含一个抽象的、可重用的 HTML 模板，所以我们将原来代码中有关列表操作的 HTML 代码提取出来，组成一个单独的 HTML 模板。每次组件创建时，Knockout 都会将组件插入页面，并将组件的视图模型绑定到 DOM 元素上。

输入列表组件需要同时支持有序和无序列表，所以模板需要使用 Knockout 绑定，以智能地区分使用哪种展现形式。两种列表都需要支持添加和删除成员，同时有序列表还要支持调整成员顺序。借助于传入的 `params` 对象，输入列表视图模型可以暴露一些布尔型属性，调整模板的展现。比方说，当 `isOrdered` 为 `true` 时，模板展示为有序列表；否则展示为无序列表。类似地，添加或删除的输入框和按钮可以使用 `enableAdd` 与 `enableRemove` 调整。

我们可以在页面中用 `<template>` 或者 `<script type="text/html">` 标签来定义组件的模板。如清单 7-29 所示，所有的标签与绑定语句都包含在一个 `<template>` 标签中。这个标签的 `id` 则用来标识模板。当我们的组件注册到 Knockout 时，程序可以通过这个 `id` 找到组件所需的 HTML 模板。

清单 7-29 Input List 组件的模板

```

<!-- example-003/public/index.html -->
<template id="item-list-template">
  <!-- ko if: isOrdered -->
  <!-- #1 THE ORDERED LIST -->
  <ol data-bind="foreach: items" class="listless">
    <li>
      <input data-bind="value: $data,
        valueUpdate: 'input',
        attr: {name: 'item-' + $index()},
        event: {input: $parent.changeItem.bind($parent, $index())}"
        type="text" />
      <button data-bind="click: $parent.demoteItem.bind($parent, $index())"
        class="fa fa-caret-down"></button>
      <button data-bind="click: $parent.promoteItem.bind($parent, $index())"
        class="fa fa-caret-up"></button>
      <button data-bind="click: $parent.removeItem.bind($parent, $index()),
        visible: $parent.enableRemove"
        class="fa fa-minus"></button>
    </li>
  </ol>
  <!-- /ko -->

  <!-- ko ifnot: isOrdered -->
  <!-- #2 THE UN-ORDERED LIST -->
  <ul data-bind="foreach: items" class="listless">
    <li>
      <input data-bind="value: $data,
        valueUpdate: 'input',
        attr: {name: 'item-' + $index()},
        event: {input: $parent.changeItem.bind($parent, $index())}"
        type="text" />
      <button data-bind="click: $parent.removeItem.bind($parent, $index()),
        visible: $parent.enableRemove"
        class="fa fa-minus"></button>
    </li>
  </ul>
  <!-- /ko -->

  <!-- ko if: enableAdd -->
  <!-- #3 THE NEW ITEM FIELD -->
  <input data-bind="value: newItem"
    type="text"
    name="new-item" />
  <button data-bind="click: commitNewItem"
    class="fa fa-plus"></button>
  <!-- /ko -->
</template>

```

input-list 组件所含的标记非常多，但实际上只是原来有序列表及无序列表的组合，并且两者共用一个新建项目的输入框。

我们用到了非常特别的绑定注释代码块：ko if 与 ko ifnot，可以根据判断条件确定注释代码块中的内容是否会出现于页面中。if 绑定与 visible 绑定的不同之处在于，visible 只是隐藏 DOM 中已经存在的元素。

提示 这种注释代码块绑定语法被称作“无容器控制流语法”。

`input-list` 模板中的显示字段与按钮都会绑定视图模型中的属性与方法。视图模型通过 `params` 的 `isOrdered` 属性来判断显示哪种列表，通过 `enableAdd` 与 `enableRemove` 来控制添加按钮与删除按钮的显示、隐藏。因为这些属性都由 `params` 传入，所以我们可以 `<input-list>` 标签上通过绑定的形式传入合适的值。通过这种方式，任何能够引起数据集改变的输入列表操作，都可以被组件抽象封装起来。

7.4.3 注册 `input-list` 组件

自定义组件在完成视图模型和模板的定义后，还需要注册到 Knockout 上，从而告知 Knockout 在遇到自定义标签后要如何实例化组件，以及使用什么模板和视图模型来渲染组件内容。

清单 7-30 中，我们修改了 `app.js` 文件，在 DOM 就绪之后立即注册了 `input-list` 组件，并且代码写在其他 Knockout 绑定的实现 (`ko.applyBindings()`) 之前。这样确保在具体的业务逻辑执行之前，我们的组件有足够的时间来初始化。

清单 7-30 注册 `input-list` 组件

```
// example-003/public/scripts/app.js
(function app ($, ko, InputList /*...*/) {
    // ...

    $(function () {
        // register the custom component tag before
        // Knockout bindings are applied to the page
        ko.components.register('input-list', {
            template: {
                element: 'item-list-template'
            },
            viewModel: InputList.create
        });

        // ...
    });

    (window.jQuery, window.ko, window.InputList /*...*/));
```

在清单 7-30 中，我们用 `ko.components.register()` 方法注册了 `input-list` 组件。这个方法接收两个参数，第一个参数是自定义组件对应的 HTML 标签的名字 `input-list`，第二个参数是对组件的一些配置。

通过对标签名字的设置，Knockout 知道 DOM 树中的 `<input-list>` 标签会被组件的模板替换，并关联一个组件的实例。

因为组件的模板定义在 `<template>` 中，所以 Knockout 只需要知道这个 `<template>` 的 ID。在组件的配置对象中，`template` 对象包含了一个 `element` 属性，属性的值就是这个 ID。对于比较简单的组件，我们可以直接将组件的模板作为字符串，直接传入 `template` 属性。

接下来看配置对象的 `viewModel` 属性。这里我们传入了一个工厂函数，当然也可以传入一个构造函数，不过使用工厂函数可以避免很多问题，如绑定了视图模型之后，函数中 `this` 指向了错误的值。创建组件的过程中，这个工厂方法会接收到一个 `params` 对象，这个对象包含了组件调用者传入的所有绑定语句。

■ **提示** 也可以通过 RequireJS 获取组件模板及视图模型方法，具体细节请参考 Knockout 的官方文档。关于 RequireJS 的介绍可以参考第 5 章。

既然我们已经注册了 `input-list` 组件，接下来就可以替换程序中关于食材列表与操作步骤列表的代码了。清单 7-31 展示的是最终更为轻量、整洁的代码。

清单 7-31 使用 Input List 组件简化代码

```

<!-- example-003/public/index.html -->

<h2>Ingredients</h2>
<!-- in read-only view -->
<ul data-bind="foreach: ingredients, visible: !isEditing()">
  <li data-bind="text: $data"></li>
</ul>
<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">
  <input-list params="items: ingredients,
    isOrdered: false"></input-list>
</div>

<h2>Instructions</h2>
<!-- in read-only view -->
<ol data-bind="foreach: instructions, visible: !isEditing()">
  <li data-bind="text: $data"></li>
</ol>
<!-- in edit view -->
<div data-bind="visible: isEditing" class="edit-field">
  <input-list params="items: instructions,
    isOrdered: true"></input-list>
</div>

```

尽管 `input-list` 组件背后拥有非常复杂的页面模式，拥有各种编辑 `input` 列表的能力，但这所有的一切最终只表现为一个简单的 `HTML` 元素，这就是组件的魅力。

7.5 Subscribable: 简单的消息传递

到现在为止，我们虽然可以修改菜谱的详情，但是并没有持久化保存这些修改，同时我们的更改也不会实时地作用到菜谱列表上。例如，当用户在菜谱详情中修改了菜谱的标题，但在列表中，用户看到的还是原来的标题。只有在用户单击 `Save` 按钮，数据成功发送到后台之后，列表才会被更新。我们需要一种更好的办法，来改进工作流程。

Knockout 提供了 `subscribable` 的概念，这是一个抽象的对象，不用于维护数据，而是提供了一种事件机制，可以被别的对象订阅。`observable` 对象可以借助 `subscribable` 提供的接口，将数据变化的消息发送给监听这个消息的对象。

`subscribable` 可以直接作为属性加到视图模型上，也可以作为一个简单的对象使用。在清单 7-32 中，我们在 `app.js` 文件中创建了一个 `subscribable` 对象，并将它作为参数传入菜谱列表及菜谱详情模块中。注意，与 `observable` 对象不同，`subscribable` 需要借助关键字 `new` 来实例化。

清单 7-32 将 Knockout Subscribable 作为一个消息总线使用

```

// example-004/public/scripts/app.js
var bus = new ko.subscribable();
var list = RecipeList.create(recipes, bus);
var details = RecipeDetails.create(list.selectedRecipe(), bus);

```

我们需要对菜谱详情的代码做一些修改，以确保菜谱更新的消息借助 `subscribable` 正确地发送出去。

首先, 这个 subscribable 被作为参数传入构造模块的工厂方法。在方法内部, 我们可以在数据发生变化时通过 subscribable 触发事件。

其次, 视图模型需要获取菜谱数据的 ID, 因为在更新菜谱的时候, 需要通过 ID 告诉服务器更新的是哪一个菜谱对象。数据更新之后, 菜谱列表也需要这个 ID 来识别哪一个菜谱对象需要更新。

最后, 我们需要修改原有的 save() 方法, 通过传入的 subscribable 对象 bus 来触发 recipe.saved 事件。具体如何修改请参考清单 7-33。

清单 7-33 在菜谱详情视图模型中保存修改的菜谱

```
// example-004/public/scripts/recipe-details.js
viewmodel.save = function () {
  var savedRecipe = {
    id: viewmodel.id,
    title: viewmodel.title(),
    ingredients: viewmodel.ingredients(),
    instructions: viewmodel.instructions(),
    cookingTime: {
      hours: viewmodel.hours(),
      minutes: viewmodel.minutes()
    },
    servings: viewmodel.servings(),
    citation: viewmodel.citation()
  };
  bus.notifySubscribers(savedRecipe, 'recipe.saved');
  viewmodel.isEditing(false);
};
```

subscribable 对象的 notifySubscribers() 方法接收两个参数: 第一个参数是即将要发送给订阅者的数据, 第二个参数是事件的名字。清单 7-34 展示的是事件 recipe.saved 的监听者对于这个事件的处理: 创建一个 AJAX 请求, 将修改后的菜谱数据发送给服务器。注意, 在 app.js 中, 所有方法共用了同一个 subscribable 对象 bus。

清单 7-34 将修改后的菜谱数据持久化到服务器

```
// example-004/public/scripts/app.js
var bus = new ko.subscribable();

bus.subscribe(function (updatedRecipe) {
  $.ajax({
    method: 'PUT',
    url: '/recipes/' + updatedRecipe.id,
    data: updatedRecipe
  }).then(function () {
    bus.notifySubscribers(updatedRecipe, 'recipe.persisted');
  })
}, null, 'recipe.saved');
```

subscribable 对象的 subscribe() 方法接收三个参数:

- 事件触发之后, 需要执行的回调函数
- 回调函数的上下文, 也就是回调函数中 this 的取值; 如果回调函数中不会用不到 this, 这个参数可以直接传 null
- 具体的事件名称, 如案例中的 recipe.saved

当该 AJAX 请求成功更新之后, 全局 subscribable 对象将会触发一个 recipe.persisted 事件; 而在菜谱列表的模块中会监听这个事件, 事件触发时, 程序会根据这个菜谱对象的 ID, 更新菜谱列

表的数据集合，并在列表中高亮这个菜谱。

清单 7-35 用保存的菜谱对象更新菜谱列表

```
// example-004/public/scripts/recipe-list.js
window.RecipeList = (function (ko) {
    return {
        create: function (recipes, bus) {
            var viewModel = {};

            // properties
            viewModel.recipes = ko.observableArray(recipes);
            viewModel.selectedRecipe = ko.observable(recipes[0]);

            // ...
            bus.subscribe(function (updatedRecipe) {

                var recipes = viewModel.recipes();
                var i = 0,
                    count = recipes.length;
                while (i < count) {
                    if (recipes[i].id !== updatedRecipe.id) {
                        i += 1;
                        continue;
                    }
                    recipes[i] = updatedRecipe;
                    viewModel.recipes(recipes);
                    viewModel.selectRecipe(recipes[i]);
                    break;
                }

                }, null, 'recipe.persisted');
            // ...
        }
    };
})(window.ko));
```

尽管 subscribable 并不是实现事件系统的唯一方法，但它确实是多模块程序中一种非常有效的解耦方法。

7.6 小结

很多前端框架都会提供一大堆令人眼花缭乱的特性和插件，而 Knockout 则专注于应用中 HTML 视图与数据模型之间的交互。Knockout 的 observable 机制极大地减轻了手动维护 HTML DOM 元素进行数据获取与更新的痛苦。开发者可以为页面上的任何元素添加 data-bind 属性，从而在该元素与一个或多个视图模型之间建立双向数据绑定。

Knockout 不仅可以将表单数据绑定到视图模型的属性，还可以将 DOM 事件绑定到视图模型的方法。在这些方法中，我们对 observable 属性的修改将会立即反映到具体 DOM 树的变化。还有更多有趣的绑定，如 visible 与 css 可以控制 DOM 元素如何显示，而 text 与 value 则可以控制元素呈现的具体内容。

AngularJS

构建大型应用的秘密是不再构建大型应用。将程序分解为容易测试的小模块，然后组装在一起就是大型应用。

—— 贾斯汀·梅耶，JavaScriptMVC 的作者

AngularJS 在开发者社区赢得了极大的关注，大批的拥趸追随其后，质疑的声音也不断增长。AngularJS 解决了许多单页应用的难题，但其解决方法与其他流行框架具有非常大的差异。

本章将介绍 AngularJS 的特点，阐述 AngularJS 适用于什么类型的项目、不适用于什么类型的项目。在本章的最后，会简单地讨论 AngularJS 的历史及其未来的发展趋势。

8.1 声明式 Web 编程

多数开发者可能更习惯“命令式编程”，而 AngularJS 则推崇“声明式编程”。尽管两种编程方式之间的区别非常微妙，但是 AngularJS 引入“声明式编程”带来了极大的方便。下面让我们通过两个案例看一下这两种编程方式。

8.1.1 命令式编程

命令式编程：传达一个命令而不是情况说明或者提出问题。

——Merriam-Webster.com

命令式编程是最好理解的编程方式，开发者向电脑发出命令，电脑会按照期望的方式执行这个命令。下面的示例中创建了一个简单的 Web 程序，展示了一个动物的无序列表。

清单 8-1 简单的命令式 Web 程序

// example-imperative/public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>命令式编程</title>
</head>
<body>
  <ul id="myList">
```



```

</ul>

<script src="/bower_components/jquery/dist/jquery.js"></script>

<script>
var App = function App() {
  this.init = function() {
    var animals = ['cats', 'dogs', 'aardvarks', 'hamsters', 'squirrels'];
    var $list = $('#myList');
    animals.forEach(function(animal) {
      $list.append('<li>' + animal + '</li>');
    });
  };
};

var app = new App();
app.init();
</script>

</body>
</html>

```

在上面例子中，我们创建了一个动物的无序列表，具体步骤如下：

1. 程序开始执行时，创建 **App** 类的实例，并调用这个实例的 **init** 方法
2. 创建一个数组，包含所有的动物（**animals**）
3. 创建无序列表 **\$list**
4. 遍历动物数组，将动物逐个加入无序列表

使用命令式编程开发程序，程序的内容就是在描述程序做了什么以及何时去做，简单地说，就是你需要告诉电脑怎么做。

8.1.2 声明式编程

声明式编程：说明情况而不是提出问题或者传达命令。——Merriam-Webster.com

声明式编程与命令式编程基本类似，但思考问题的方式完全不同。开发者采用声明式编程，重点在于描述程序的结果，而如何达到这个结果由程序来处理。

让我们来看下面的例子，这个例子与前面的例子极为类似。在这个例子中，我们借助 AngularJS，使用声明式编程的方式创建了动物的无序列表。

清单 8-2 借助 Angular 实现声明式 Web 程序
// example-declarative/public/index.html

```

<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>声明式编程</title>
</head>
<body>

  <div ng-controller="BodyController">
    <ul>
      <li ng-repeat="animal in animals">{{animal}}</li>
    </ul>
  </div>

```

```

<script src="/bower_components/angularjs/angular.js"></script>

<script>
var app = angular.module('app', []);
app.controller('BodyController', function($scope) {
    $scope.animals = ['cats', 'dogs', 'aardvarks', 'hamsters', 'squirrels'];
});
</script>

</body>
</html>

```

上面的例子中，有很多重要的地方。不过此刻，我们先注意一下代码中，有很多 HTML 标签的属性不是标准的属性（如 `ng-app`、`ng-controller`、`ng-repeat`）。这些属性示范了 Angular 中核心概念——指令的用法。

简单地说，开发者可以通过 Angular 的指令扩展 HTML 的语法。我们很快就会看到，指令可以以 CSS 的 `class`、自定义属性、注释甚至是自定义标签的形式出现。当 Angular 发现代码中的指令，它会执行与指令相关联的代码。这个过程通常包含方法的执行、指令模板的加载等一系列复杂的工作。AngularJS 提供了很多内置的指令，如上面的示例中用到的指令。我们将在后续的文章中谈及很多内置的 Angular 指令。

当采用声明式编程方式开发 Web 程序时，程序控制流的解析工作由代码转换到界面上，我们不再需要描述程序从启动开始需要做的每一件事，转而让界面去描述将要发生的一切。Angular 指令让这一切变为可能。

8.2 模块：构建松散耦合程序的基石

当我们把复杂的程序分解为相互独立但又相互关联的组件之后，程序就变得不再复杂。在 Angular 中，我们可以通过 Angular 项目的基本单位——模块（Module）来实现上述过程。

让我们再来看一下清单 8-2，可以看到在程序中通过调用 Angular 的 `module()` 方法创建一个模块，而且整个程序只有这一个模块。`module()` 方法既可以用来创建模块，也可以用来获取模块。当我们创建一个模块时，需要传入两个参数：第一个参数是模块的名字，第二个参数是一个字符串数组，代表该模块依赖的其他模块。本例中，我们的模块没有依赖其他模块，但是对于第二个参数，我们依旧会传一个空数组，这是为了与 `module` 方法获取模块的功能加以区分。清单 8-3 展示了如何创建一个有两个依赖项的模块。

清单 8-3 创建有依赖 module 的 Angular module

```

/**
 * Creates a new module that depends on two other modules - 'module1' and 'module2'
 */
var app = angular.module('app', ['module1', 'module2']);

```

模块定义完毕之后，就可以通过 `module()` 方法的 `getter` 语义获取其引用。

清单 8-4 不传入依赖 module 数组时，Angular 的 `module` 方法返回已创建的 module

```

/**
 * Returns a reference to a pre-existing module named 'app'
 */
var app = angular.module('app');

```

在这一章, 我们会看到很多 Angular 提供的工具。你可能会逐渐意识到, 这些工具都是以某个模块作为运行环境的。每个 Angular 应用本身就是一个依赖其他模块进行工作的模块。当我们拥有模块相互依赖的概念之后, 我们就可以用可视化的形式展示一个 Angular 程序的模式了, 如图 8-1 所示。

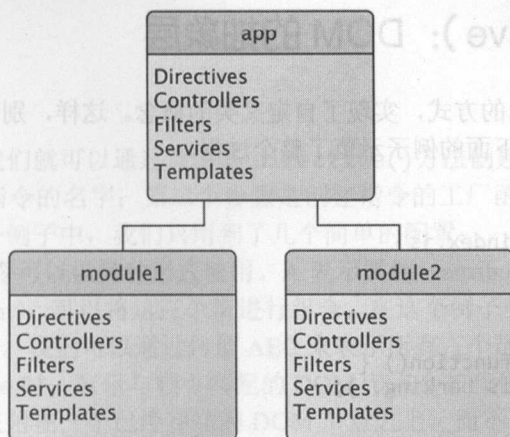


图 8-1 每个 Angular 程序都是一个 module, 并依赖其他的 module

创建引导 Module

现实世界中, 重要建筑工程始于基石的奠定。相似地, 每一个 Angular 项目也有自己的基石, 那就是一个可以代表程序本身的模块。对这个模块 (及其依赖项) 进行初始化的过程, 就称作 Angular 的“引导” (Bootstrap) 过程。通常, Angular 的引导过程可以是下面两种情况中的一种。

(1) 自动引导

让我们再来看一下清单 8-2, 注意 `html` 标签中的 `ng-app` 指令。页面加载完毕后, Angular 就会在页面中寻找 `ng-app` 指令。如果找到 `ng-app` 对应的模块, 该模块就被认定是程序的主模块。这个模块会被自动初始化, 同时意味着这个 Angular 程序已经就绪。

(2) 手动引导

对大多数 Angular 程序来说, 上面说到的自动引导已经足够。然而在某些情况下, 能够对这个引导过程加以控制, 也是非常有用的。我们将在下面的例子中演示如何手动控制这一引导过程。

清单 8-5 ajax 请求结束之后启动 Angular 程序

```

$.ajax({
  'url': '/api/data',
  'type': 'GET'
}).done(function() {
  angular.bootstrap(document, ['app']);
});
  
```

在这个示例中, AJAX 请求结束之后, 我们通过 `bootstrap()` 方法启动 Angular 程序。这个方法需要两个参数: 第一个参数是一个 DOM 节点, 也就是 Angular 程序的容器; 第二个参数是一个字符串数组, 指定这个 Angular 程序的主模块。

注意 多数情况下，一个页面中只有一个 Angular 程序。当然，Angular 也支持在一个页面中有多个 Angular 程序共存。在这种情况下，只有一个 Angular 程序可以自动引导，其他程序必须手动引导。

8.3 指令 (Directive): DOM 的抽象层

JavaScript 通过原型继承的方式，实现了自定义类的概念。这样，别的开发者可以使用这些类，而不用考虑其内部的实现。下面的例子示范了整个过程。

清单 8-6 原型继承

// example-prototype/index.js

```
function Dog() {
}

Dog.prototype.bark = function() {
  console.log('Dog is barking.');
```

```
};

Dog.prototype.wag = function() {
  console.log('Dog is wagging.');
```

```
};

Dog.prototype.run = function() {
  console.log('Dog is running.');
```

```
};

var dog = new Dog();
dog.bark();
dog.wag();
dog.run();
```

上例中，这种将复杂行为抽象为简单接口的方式，是面向对象编程思想的充分体现。相似地，Angular 指令可以看作对 DOM 的一种抽象。借助指令，开发者只要操作简单的 HTML 标签，就可以创建复杂的 Web 组件。清单 8-7 中提供的示例应该可以阐述清楚这一概念。

清单 8-7 创建一个简单的 Angular 指令

// example-directive1/public/index.html

```
<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Example Directive</title>
  <link rel="stylesheet" href="/css/style.css">
  <link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

  <news-list></news-list>

  <script src="/bower_components/angularjs/angular.js"></script>
  <script>
    var app = angular.module('app', []);
    app.directive('newsList', function() {
```



```

return {
  'restrict': 'E',
  'replace': true,
  'templateURL', '/templates/news-list.html'
};
});
</script>

```

```

</body>
</html>

```

模块创建完毕之后，我们就可以通过调用其 `directive()` 方法创建新的指令了。这个方法需要两个参数：第一个参数是指令的名字；第二个参数是创建指令的工厂函数，该方法返回一个用于创建指令的配置对象。在这个例子中，我们只用到了几个简单的配置：

restrict: 指定该指令可以以何种形式使用。A 表示属性 (attribute)，C 表示 CSS 类 (class)，E 表示 HTML 元素 (element)。可以将这三个值进行组合。在这个例子中，E 表示只可以通过标签名字来匹配指令与 DOM 元素。我们可以通过传值 AEC 来表示所有三个形式。

replace: 当值为 `true` 时，表示与指令匹配的 DOM 节点会被我们的组件替换。当值为 `false` 时，则表示我们的指令会依附在一个已经存在的 DOM 节点之上，而不是完全替换它。

templateUrl: 当我们的指令插入 DOM 树时，Angular 会通过该 URL 获取这个组件的 HTML 模板。如果希望直接传入模板的内容，可以使用 `template` 选项。

注意 请注意程序中指令的命名。当我们在 JavaScript 中创建指令时，采用的是“驼峰式”的命名格式；而在 HTML 中，采用的是“横线 (dash) 分隔”的方式。这是因为 HTML 不区分大小写。AngularJS 在编译 HTML 时会自动处理这种命名约定之间的差异。

现在，我们在浏览器中执行这个程序，Angular 会自动寻找与这个指令对应的 DOM 节点，并将每个匹配的 DOM 元素替换为指令的模板。最终效果如图 8-2 所示。

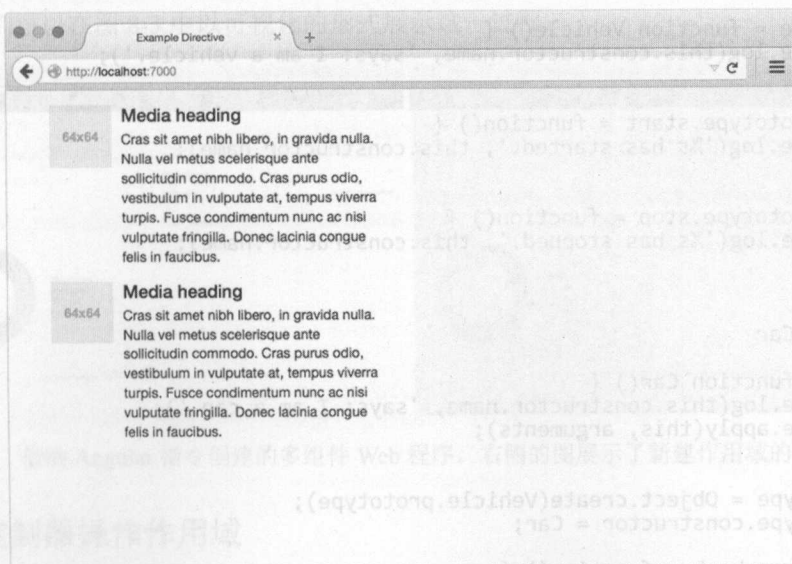


图 8-2 新创建的指令

这个简单的示例仅仅实现了使用一个模板替换 DOM 节点(我们会在后面的章节为这个指令加入自定义逻辑)。然而,读者应该已经注意到 Angular 指令带给开发者的便利与强大的功能。正如清单 8-7 所示,我们仅使用简单的标签,就为程序插入了一个复杂的组件。这种将复杂组件逻辑抽象为简单调用的方式,更加易于管理。

8.4 加入逻辑

在前面的章节,我们简单了解了 Angular 指令,并在最后的部分创建了一个简单的指令,该指令使用我们提供的模板替换了已知的 DOM 元素。为了更进一步展示指令的强大功能,我们将为这个指令加入自己的业务逻辑。为了完成这一过程,我们需要用到作用域(scope)与控制器(controller)。

8.4.1 作用域与原型继承

Angular 的作用域乍一看可能有些难以理解,这大概是因为作用域与 JavaScript 中比较难理解的部分——原型继承——关系密切。对于刚接触 Angular 的开发者来说,作用域可能是比较难理解的概念,但是充分理解作用域有利于更好地使用 Angular。在继续之前,我们先花一些时间来看一下作用域存在的意义及其工作原理。

在大多数基于“类”的面向对象语言中,继承可以通过类的语法来实现;而在 JavaScript 中,继承是通过“原型继承”来实现的。这个过程借助对象与方法来实现。清单 8-8 展示了如何实现原型继承。

清单 8-8 演示原型继承, Car 继承自 Vehicle

```
// example-prototype2/index.js

/**
 * @class Vehicle
 */
var Vehicle = function Vehicle() {
  console.log(this.constructor.name, 'says: I am a vehicle.');
```

```
};

Vehicle.prototype.start = function() {
  console.log('%s has started.', this.constructor.name);
};

Vehicle.prototype.stop = function() {
  console.log('%s has stopped.', this.constructor.name);
};

/**
 * @class Car
 */
var Car = function Car() {
  console.log(this.constructor.name, 'says: I am a car.');
```

```
  Vehicle.apply(this, arguments);
};

Car.prototype = Object.create(Vehicle.prototype);
Car.prototype.constructor = Car;

Car.prototype.honk = function() {
  console.log('%s has honked.', this.constructor.name);
```

```

});

var vehicle = new Vehicle();
vehicle.start();
vehicle.stop();

var car = new Car();
car.start();
car.honk();
car.stop();

/* Result:
Vehicle says: I am a vehicle.
Vehicle has started.
Vehicle has stopped.
Car says: I am a car.
Car says: I am a vehicle.
Car has started.
Car has honked.
Car has stopped.
*/

```

本示例中，我们首先定义了 `Vehicle` 函数，随后通过扩展 `Vehicle` 的 prototype 为 `Vehicle` 添加了两个实例方法：`start()` 与 `stop()`。我们通过 `Object.create()` 方法创建了一个对象，该对象的 prototype 指向 `Vehicle` 的 prototype。随后，我们定义了一个 `Car()` 方法，将这个 `Car` 的 prototype 指向之前 `Object.create` 方法的返回值。最后，我们为 `Car` 添加了一个实例方法 `honk()`。运行这个程序，可以发现 `Vehicle` 的实例可以调用 `start()` 与 `stop()` 方法，而 `Car` 的实例可以执行 `start()`、`stop()` 以及 `honk()`。这就是原型继承的原理。

在 Angular 中，相似的过程也在发生。在 Angular 程序的引导阶段，Angular 会创建一个“根对象”（即 `$rootScope`），这个对象绑定在 Angular 程序的根元素之上。随后，Angular 会继续遍历 DOM 树，并创建与之关联的 directive 的实例（Angular 中，这个过程称为“编译”）。在创建这些实例的过程中，绑定在 DOM 上的对象会继承自与它最近的父级对象。Angular 创建的这些对象，也就是所谓的作用域。我们可以在图 8-3 中以可视化的形式展示这一过程。

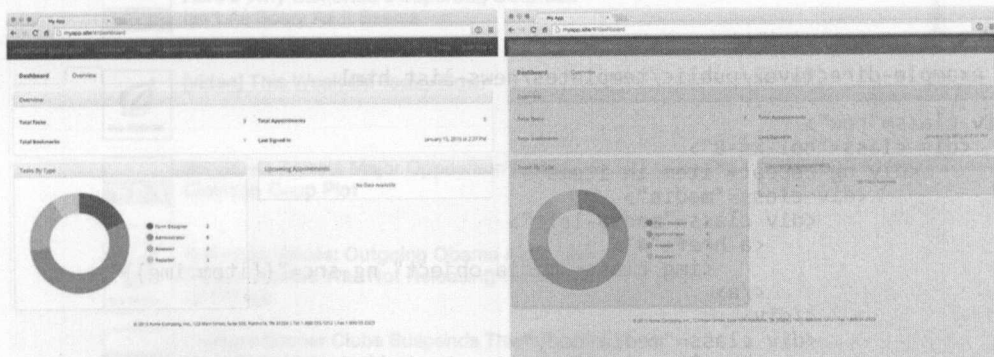


图 8-3 借助 Angular 指令创建的多组件 Web 程序，右侧的图展示了新建作用域的作用范围

8.4.2 用控制器操作作用域

Angular 控制器（Controller）其实就是一个函数，该函数存在的唯一目的就是操作作用域对象。

现在我们可以为指令加入业务逻辑了。清单 8-9 是清单 8-7 的扩展。与清单 8-7 不同的是，我们在 `directive` 方法返回的描述对象中加入了 `controller` 属性。该指令的模板如清单 8-10 所示。

清单 8-9 清单 8-7 的扩展，加入自定义行为

// example-directive2/public/index.html

```
<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Example Directive</title>
  <link rel="stylesheet" href="/css/style.css">
  <link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

  <news-list></news-list>

  <script src="/bower_components/angularjs/angular.js"></script>
  <script>
    var app = angular.module('app', []);
    app.directive('newsList', function() {
      return {
        'restrict': 'E',
        'replace': true,
        'controller': function($scope, $http) {
          $http.get('/api/news').then(function(result) {
            $scope.items = result.data;
          });
        },
        'templateUrl': '/templates/news-list.html'
      };
    });
  </script>
</body>
</html>
```

清单 8-10 directive 的模板

// example-directive2/public/templates/news-list.html

```
<div class="row">
  <div class="col-xs-8">
    <div ng-repeat="item in items">
      <div class="media">
        <div class="media-left">
          <a href="#">
            
          </a>
        </div>
        <div class="media-body">
          <h4 class="media-heading" ng-bind="item.title"></h4>
        </div>
      </div>
    </div>
  </div>
</div>
```

在清单 8-9 中，我们重点看 `directive` 方法返回对象中的 `controller` 属性。这个属性支持传

入一个方法，我们为这个方法指定了两个参数：`$scope` 与 `$http`。现阶段，先不用关心这两个参数是如何传入的，我们会在后续有关服务（service）的章节中详细讨论。如前所述，在编译与 DOM 绑定的指令时，Angular 会创建一个作用域对象；而在上面的控制器中，我们可以通过 `$scope` 拿到这个对象。Angular 支持双向数据绑定，当我们修改 `$scope` 对象时，相应的 DOM 标签也会跟着发生改变。

双向数据绑定

Angular 通过数据绑定的形式将 JavaScript 对象与 HTML 模板连接起来。借助作用域，模板依照 JavaScript 对象的描述渲染到浏览器上。图 8-4 描述了这一过程。

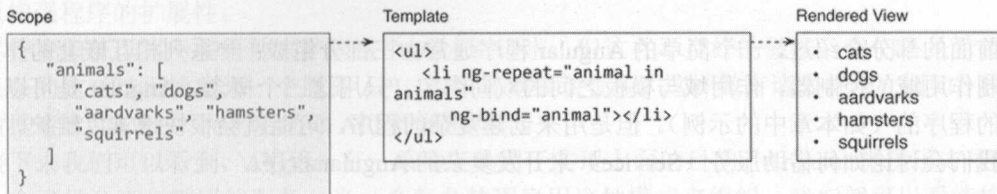


图 8-4 Angular 渲染 scope 中数据的过程

上面的例子中，`$scope` 中的数据被渲染在页面上。当然，Angular 指令不仅支持这种“单向”的数据绑定，还支持反向的过程。当页面上的内容发生改变时，与之绑定的数据也会发生变化，这也就是所谓的“双向”数据绑定。

注意 在后面“创建复杂表单”部分，我们会对双向数据绑定做更深入的讨论。

在清单 8-9 中，借助 Angular 的 `$http` 服务，我们获取了 National Public Radio 以及 The Onion

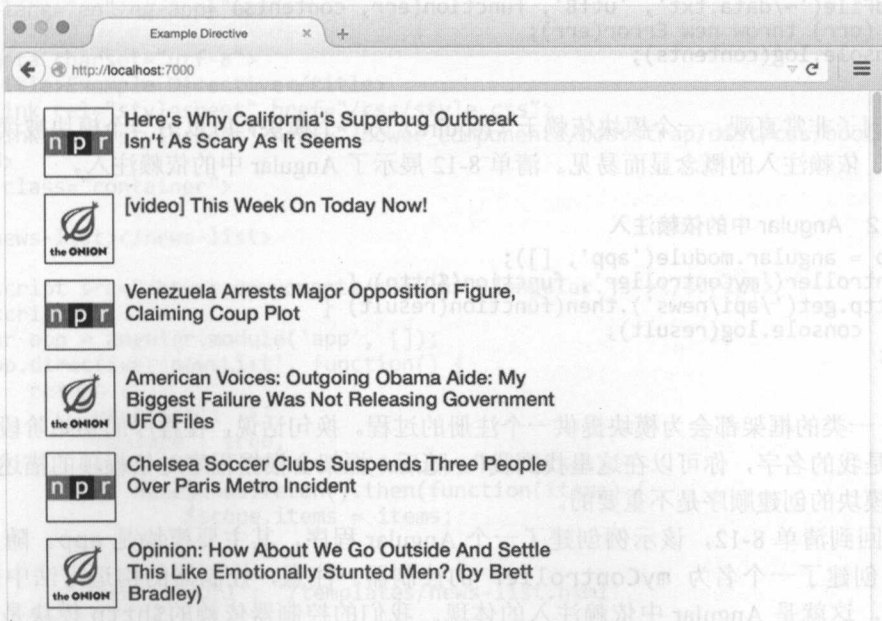


图 8-5 程序的最终效果

的头条新闻，请求的结果传入了 `$scope` 的 `items` 数组。至于页面如何展示这个数组，我们需要看清单 8-10 中的 HTML 模板。这个模板中用到了比较重要的 Angular 指令 `ng-repeat`，它会为 `items` 中的每一个元素创建一个 `<div class="media">...</div>` 标签。随后，程序借助 Angular 内置指令 `ng-src` 与 `ng-bind`，按照模板提供的方式展示新闻的图片以及文字内容。

程序最终的展示效果如图 8-5 所示。

8.5 通过服务与依赖注入实现松散耦合

在前面的部分介绍过，一个简单的 Angular 程序通常由三部分组成：一系列相互嵌套的作用域对象、控制作用域的控制器、作用域与模板之间的双向绑定。只用这三个概念，Angular 是可以创建一些简单的程序的（如本章中的示例）。但是用来创建复杂的程序，可能就会很快地难以维护。在这一部分，我们会讨论如何借助服务（Service）来开发复杂的 Angular 程序。

8.5.1 依赖注入（Dependency Injection）

介绍服务之前，我们先来简单看一下依赖注入的概念。这个概念在客户端框架中较少被提及，但在 Angular 中却十分常见。

首先，我们来看一下清单 8-11。这个例子中展示的是一个简单的 Node 应用，该应用依赖于 `fs` 模块，我们的程序通过 `require()` 方法获取了 `fs` 模块。

清单 8-11 依赖 `fs` 模块的 Node.js 程序

```
var fs = require('fs');
fs.readFile('~/data.txt', 'utf8', function(err, contents) {
  if (err) throw new Error(err);
  console.log(contents);
});
```

上面的例子非常直观，一个模块依赖于（`require`）另一个模块，所以另一个模块被读取并返回给第一个模块。依赖注入的概念显而易见。清单 8-12 展示了 Angular 中的依赖注入。

清单 8-12 Angular 中的依赖注入

```
var app = angular.module('app', []);
app.controller('myController', function($http) {
  $http.get('/api/news').then(function(result) {
    console.log(result);
  });
});
```

Angular 一类的框架都会为模块提供一个注册的过程。换句话说，在程序的初始阶段，模块会告诉程序“这是我的名字，你可以在这里找到我”。随后，框架会根据程序对依赖项的描述，自动加载这些模块。模块的创建顺序是不重要的。

让我们回到清单 8-12，该示例创建了一个 Angular 程序，其主要模块是 `app`。随后，我们在 `app` 模块下创建了一个名为 `myController` 的控制器。注意，控制器的构造方法中传入了一个 `$http` 参数，这就是 Angular 中依赖注入的体现。我们的控制器依赖的 `$http` 模块是 Angular 的内置模块。在程序的引导阶段，Angular 将这个模块注册为服务。我们将在后面的部分讲述如何

创建一个服务。

■ **注意** 习惯上, Angular 内置的 service、API 以及属性都会加前缀。

8.5.2 简单的控制器与复杂的服务

让我们再回顾一下清单 8-9, 该示例展示了如何为一个指令添加业务逻辑。代码中, 我们发送了一个 AJAX 请求, 以获取一些新闻头条。然而, 这个例子还需要缓存请求的数据, 在整个程序中共享, 以增强程序的扩展性。

尽管我们可以写一些组件来扩展这个 AJAX 请求, 但这并不是好的做法。我们还是希望将获取新闻头条的这个过程抽象为一个服务 API, 以供其他模块使用。这样做有诸多优点, 例如, 我们可以修改数据请求的 URL, 而不用通知 API 的调用者。

接下来我们可以看到, Angular 有一系列工具用于实现上面的目的。Angular 提供了方便的途径, 创建一个可供分享与重用的服务。当一个复杂的程序用这种模式来组织, 我们就可以看清控制器的实质: 控制器实际上就是作用域与服务之间的胶水层, 共同为界面的渲染服务。

Angular 提供了三种创建服务的方式: factory、service 以及 provider。接下来分别介绍这几种方式。

1. Factory

清单 8-13 用 factory 方法创建 service

// example-directive3/public/index.html

```
<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Example Directive</title>
  <link rel="stylesheet" href="/css/style.css">
  <link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

  <news-list></news-list>

  <script src="/bower_components/angularjs/angular.js"></script>
  <script>
    var app = angular.module('app', []);
    app.directive('newsList', function() {
      return {
        'restrict': 'E',
        'replace': true,
        'controller': function($scope, headlines) {
          headlines.fetch().then(function(items) {
            $scope.items = items;
          });
        },
        'templateUrl': '/templates/news-list.html'
      };
    });
  </script>
  app.factory('headlines', function($http) {
```

```

    return {
      'fetch': function() {
        return $http.get('/api/news').then(function(result) {
          return result.data;
        });
      }
    };
  });
</script>
</body>
</html>

```

在清单 8-13 中, **headlines** 工厂函数返回了一个包含 **fetch()** 方法的对象。这个方法被调用时, 程序会请求数据, 并将结果以 **promise** 对象的形式返回。

在大多数 Angular 程序中, 工厂函数 **factory** 是创建服务最常见的方式。当一个 **factory** 作为依赖项第一次出现时, Angular 会调用这个 **factory** 方法, 并返回方法的执行结果。随后对于 **factory** 的调用, 返回的都是第一次调用的结果。换句话说, **factory** 是单例的, 只会执行一次。

2. Service

清单 8-14 用 **service** 方法创建 **service**

// example-directive4/public/index.html

```

<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Example Directive</title>
  <link rel="stylesheet" href="/css/style.css">
  <link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

  <news-list></news-list>

  <script src="/bower_components/angularjs/angular.js"></script>
  <script>
    var app = angular.module('app', []);
    app.directive('newsList', function() {
      return {
        'restrict': 'E',
        'replace': true,
        'controller': function($scope, headlines) {
          headlines.fetch().then(function(items) {
            $scope.items = items;
          });
        },
        'templateUrl': '/templates/news-list.html'
      };
    });
    app.service('headlines', function($http) {
      this.fetch = function() {
        return $http.get('/api/news').then(function(result) {
          return result.data;
        });
      };
    });
  </script>

```



```
</script>
```

```
</body>
</html>
```

在 Angular 中, service 方法与 factory 方法几乎一模一样的。不同的地方是, factory 方法会被 Angular 直接调用;而 service 方法则被看作 service 的构造方法,会通过 new 的方式创建服务,就像在 JavaScript 中创建类的实例一样。使用 factory、service 两种方案中的哪一种完全取决于你的喜好,两者的结果是完全相同的。

在这个例子中,不同于在 factory 方法中返回一个对象,我们在 service 方法中为 this 添加了一个 fetch() 方法,而 service 对象则由这个构造方法创建。

3. Provider

清单 8-15 用 provider 方法创建 service

// example-directive5/public/index.html

```
<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Example Directive</title>
  <link rel="stylesheet" href="/css/style.css">
  <link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

  <news-list></news-list>

  <script src="/bower_components/angularjs/angular.js"></script>
  <script>
    var app = angular.module('app', []);
    app.directive('newsList', function() {
      return {
        'restrict': 'E',
        'replace': true,
        'controller': function($scope, headlines) {
          headlines.fetch().then(function(items) {
            $scope.items = items;
          });
        },
        'templateUrl': '/templates/news-list.html'
      };
    });
    app.config(function(headlinesProvider) {
      headlinesProvider.limit = 10;
    });
    app.provider('headlines', function() {
      this.$get = function($http) {
        var self = this;
        return {
          'fetch': function() {
            return $http.get('/api/news', {
              'params': {
                'limit': self.limit || 20
              }
            }).then(function(result) {
              return result.data;
            });
          }
        };
      };
    });
  </script>
</body>
```

```

    });
  });
});
</script>
</body>
</html>

```

与 `factory` 和 `service` 不同, `provider` 允许开发者在其所属 `module` 的初始阶段 (`module` 的 `config` 方法的执行阶段), 就对其进行配置。可以认为 `provider` 是可配置的 `factory`。在上面的示例代码中, 我们定义了一个叫作 `headlines` 的 `provider`, 其作用类似于清单 8-13。不同的地方在于, 这里提供了一个 `limit` 参数, 限制请求结果的条数。

在清单 8-15 中, 我们在 `provider` 方法中, 通过 `this.$get` 定义了一个 `factory` 方法。当 `headlines` 作为依赖项使用时, Angular 会将这个 `provider` 方法的执行结果返回, 类似于清单 8-13 中 `factory` 的效果。与之对照, 请注意一下代码中的 `fetch()` 方法, 重点看 `config` 方法中配置的 `limit` 参数如何影响程序的执行效果。

8.6 创建路由

通过 Angular 等框架开发的单页应用 (Single Page Application, SPA), 给用户提供了类似于传统桌面应用的流畅体验。要实现这种效果, 程序所需的大部分资源 (脚本、样式表等) 应该在程序的唯一页面中提前加载, 而后续的请求则通过 AJAX 实现, 以防页面刷新。在这一部分, 你将学到如何使用 `ngRoute` 模块来实现单页面的管理。

我们将在清单 8-16 中, 我们继续扩展清单 8-13。这一次, 清单 8-16 为程序加入两个路由项, 引导用户访问两个不同的部分: “Dashboard (仪表器)” 与 “News Headlines (新头条)”。当用户访问 `/#/headlines` 时, `newsList` 指令才会被注入页面。为此, 以下步骤:

1. 在程序的引导阶段, 执行主模块的 `config` 方法。而在 `config` 方法中, 我们获取了 `$routeProvider` 用于配置。注意, 在 `app` 模块中引入 `ngRoute` 模块依赖。
2. 在 `config` 方法中, 定义一个 `routes` 数组用于存储程序的路由信息。在本示例中, 每个对象的 `route` 属性定义了路由的访问地址, `config` 属性允许我们指定路由的 `controller` 方法以及加载模板的路径。
3. 遍历 `routes` 数组的每一项, 将合适的属性传入 `$routeProvider` 的 `when()` 方法。这提供了一种简单配置多路由的方案。其实, 在下面的例子中, 我们可以不用数组, 而是简单地调用两次 `$routeProvider.when()` 方法。
4. 最后, 用 `$routeProvider.otherwise()` 方法定义默认路由, 用于处理用户没有输入路由或者输错路由地址的情况。

清单 8-16 定义 `dashboard` 与 `headlines` 两个路由

```

// example-router1/public/index.html
<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Routing Example</title>

```

```

<link rel="stylesheet" href="/css/style.css">
<link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

  <ng-view></ng-view>

  <script src="/bower_components/angularjs/angular.js"></script>
  <script src="/bower_components/angular-route/angular-route.js"></script>
  <script src="/modules/news-list.js"></script>
  </script>
  var app = angular.module('app', ['ngRoute', 'newsList']);
  app.config(function($routeProvider) {
    var routes = [
      {
        'route': '/dashboard',
        'config': {
          'templateUrl': '/templates/dashboard.html'
        }
      },
      {
        'route': '/headlines',
        'config': {
          'controller': function($log) {
            $log.debug('Welcome to the headlines route.');
          },
          'templateUrl': '/templates/headlines.html'
        }
      }
    ];
    routes.forEach(function(route) {
      $routeProvider.when(route.route, route.config);
    });
    $routeProvider.otherwise({
      'redirectTo': '/dashboard' // 默认路由
    });
  });
</script>

</body>
</html>

```

8.6.1 路由参数

通常来说，在一个 Angular 程序中，路由参数不同，页面最终的表现也不一样。清单 8-17 展示了这一现象。

清单 8-17 需要参数的路由配置

// example-router2/public/index.html

```

<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Routing Example</title>
  <link rel="stylesheet" href="/css/style.css">
  <link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

```

```

<ng-view></ng-view>
<script src="/bower_components/angularjs/angular.js"></script>
<script src="/bower_components/angular-route/angular-route.js"></script>
<script>
var app = angular.module('app', ['ngRoute']);
app.config(function($routeProvider) {
  var routes = [{
    'route': '/dashboard',
    'config': {
      'templateUrl': '/templates/dashboard.html',
      'controller': function($scope, $http) {
        return $http.get('/api/animals').then(function(result) {
          $scope.animals = result.data;
        });
      }
    }
  ],
  'route': '/animals/:animalID',
  'config': {
    'templateUrl': '/templates/animal.html',
    'controller': function($scope, $route, $http) {
      $http.get('/api/animals/' + $route.current.params.animalID).
    then(function(result) {
      $scope.animal = result.data;
    });
  }
  });
  routes.forEach(function(route) {
    $routeProvider.when(route.route, route.config);
  });
  $routeProvider.otherwise({
    'redirectTo': '/dashboard' // Our default route
  });
});
</script>
</body>
</html>

```

8.6.2 路由的 Resolve

随着用户需求的提升，程序的功能也变得越来越复杂。然而，这种改变不是没有代价的。例如，在一个单页面应用的生命周期中，协调各个请求就是一件困难的事。在继续之前，我们先来看一下 Angular 的 `ngRoute` 提供的一个有用的功能：`resolve`，它能帮我们解决前面说到的复杂情况。

`Resolve` 可以帮助我们实现这样的效果：等一些步骤完成之后，路由才会发生切换。如果这些步骤返回的是 `promise` 对象，那么必须等到每个 `promise` 对象都 `resolve` 之后，路由才会触发切换。清单 8-18 展示了路由的 `resolve`。

清单 8-18 路由的 `Resolve`

// example-router3/public/index.html

```

<!DOCTYPE html>
<html lang="en" ng-app="app">

```



```

<head>
  <meta charset="utf-8">
  <title>Routing Example</title>
  <link rel="stylesheet" href="/css/style.css">
  <link rel="stylesheet" href="/bower_components/bootstrap/dist/css/bootstrap.css">
</head>
<body class="container">

  <ng-view></ng-view>

  <script src="/bower_components/angularjs/angular.js"></script>
  <script src="/bower_components/angular-route/angular-route.js"></script>
  <script>
    var app = angular.module('app', ['ngRoute']);
    app.config(function($routeProvider) {
      $routeProvider.when('/dashboard', {
        'templateUrl': '/templates/dashboard.html',
        'controller': function($scope, animals, colors) {
          $scope.animals = animals;
          $scope.colors = colors;
        },
        'resolve': {
          'animals': function($http) {
            return $http.get('/api/animals').then(function(result) {
              return result.data;
            });
          },
          'colors': function($http) {
            return $http.get('/api/colors').then(function(result) {
              return result.data;
            });
          }
        }
      });
      $routeProvider.otherwise({
        'redirectTo': '/dashboard' // Our default route
      });
    });
  </script>

</body>
</html>

```

在上面的例子中，我们定义了一个路由，对应的页面展示了 **animals** 列表与 **colors** 列表。这两个列表的数据需要从服务器获取。我们没有在 **controller** 方法中直接请求这些数据，而是将其写在了路由对象的 **resolve** 对象中。这样，当程序执行到 **controller** 方法中时，我们可以确定，这些数据已经准备好了。

8.7 创建复杂表单

HTML 表单的管理十分困难，主要的问题就是表单的验证。所谓验证，就是告诉用户发生了什么错误（如必填的字段没有填写），并引导用户解决问题。另外，复杂表单的内容通常是可变的，根据用户的输入不同，界面的表现也会不同。在接下来的章节，我们会通过一系列的示例来说明 Angular 在表单处理方面的优势。

8.7.1 表单验证

设计良好的表单通常比较重视用户体验。这些表单不会假定用户完全理解表单的填写要求，而是当出现问题时，告知用户如何解决这个问题。幸运的是，Angular 的特殊语法可以帮助开发者简单地创建如上所述的表单。

清单 8-19 展示了第一个例子的 HTML 代码，与之相应的控制器则展示在清单 8-20 中。

清单 8-19 表单验证以及表单的动态反馈

// example-form1/public/index.html

```
<!DOCTYPE html>
<html lang="en" ng-app="app">
<head>
  <meta charset="utf-8">
  <title>Example Form</title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body ng-controller="formController">

  <form name="myForm" ng-class="formClass" ng-submit="submit()" novalidate>

    <div class="row">

      <div ng-class="{
        'has-error': !myForm.first_name.$pristine && !myForm.first_name.$valid,
        'has-success': !myForm.first_name.$pristine && myForm.first_name.$valid
      }">

        <label>First Name</label>
        <input
          type="text"
          name="first_name"
          ng-model="model.first_name"
          class="form-control"
          ng-minlength="3"
          ng-maxlength="15"
          ng-required="true">

        <p ng-show="
          !myForm.first_name.$pristine &&
          myForm.first_name.$error.required">
          First name is required.

        </p>
        <p ng-show="
          !myForm.first_name.$pristine &&
          myForm.first_name.$error.minlength">
          First name must be at least 3 characters long.

        </p>
        <p ng-show="
          !myForm.first_name.$pristine &&
          myForm.first_name.$error.maxlength">
          First name can have no more than 15 characters.

        </p>
      </div>

      <div ng-class="{
        'has-error': !myForm.last_name.$pristine && !myForm.last_name.$valid,
```

```

'has-success': !myForm.last_name.$pristine && myForm.last_name.$valid
}">
<label>Last Name</label>
<input
  type="text"
  name="last_name"
  ng-model="model.last_name"
  class="form-control"
  ng-minlength="3"
  ng-maxlength="15"
  ng-required="true">
<p ng-show="
  !myForm.last_name.$pristine &&
  myForm.last_name.$error.required">
  Last name is required.
</p>
<p ng-show="
  !myForm.last_name.$pristine &&
  myForm.last_name.$error.minlength">
  Last name must be at least 3 characters long.
</p>
<p ng-show="
  !myForm.last_name.$pristine &&
  myForm.last_name.$error.maxlength">
  Last name can have no more than 15 characters.
</p>
</div>
</div>

<div class="row">
  <div>
    <button type="submit" ng-disabled="myForm.$invalid">Submit</button>
    <button type="button" ng-click="reset()">Reset</button>
  </div>
</div>
</form>

<hr>

<div class="output" ng-bind="output"></div>

<script src="/bower_components/angularjs/angular.js"></script>
<script src="/app/index.js"></script>

</body>
</html>

```

清单 8-20 绑定在 body 元素上的 controller 方法

// example-form1/public/app/index.js

```

var app = angular.module('app', []);
app.controller('formController', function($scope, $http, $log) {

  $scope.formClass = null;
  $scope.model = {};

  $http.get('/api/model').then(function(result) {
    $scope.model = result.data;
  });
});

```

```

});

$scope.submit = function() {
    if (!$scope.myForm.$valid) return;
    $http.post('/api/model', {
        'model': $scope.model
    }).then(function() {
        alert('Form submitted.');
```

设计时，这些表单不会创建，而是当出现问题时，才会创建如上所示的模板代码。

```

    }).catch(function(err) {
        alert(err);
    });
};

$scope.reset = function() {
    $scope.model = {};
    $http.post('/api/model', {
        'model': $scope.model
    });
};

/**
 * Angular 内置的 '$watch()' 方法
 * 允许我们监听 $scope 对象属性的变化
 * 这里将表单的内容保存为一个 JSON 字符串
 * 并将其赋值给 $scope.output
 */
$scope.$watch('model', function() {
    $scope.output = angular.toJson($scope.model, 4);
}, true);
});

```

Angular 内置了一个控制器，用于完成表单相关的操作，这个控制器就是 `FormController`。当 Angular 编译我们的程序时，模板中的 `<form>` 标签会被绑定一个内置的 `form` 指令。在这个指令中，程序会创建 `FormController` 的实例。最后，根据 `<form>` 标签 `name` 属性的值（我们的例子中是 `myForm`），Angular 会在 `<form>` 所属作用域上添加对这个 `FormController` 实例的引用。如上面的例子，我们就可以在控制器中通过 `$scope.myForm` 获取这个 `FormController` 的实例。

`FormController` 提供了很多有用的属性与方法。我们可以在清单 8-19 以及清单 8-20 中看到一些示例。首先，可以看到，通过 `ng-disabled`，我们可以动态切换表单提交按钮的可用与不可用状态。在上述示例中，我们将表单的 `$invalid` 属性绑定在 `ng-disabled` 上。`$invalid` 取值 `TRUE` 或者 `FALSE`，表示表单中的 `input` 是否都通过了验证。

在清单 8-19 中，我们为表单添加了更多的指令，用于简单的验证规则，如 `ng-minlength`、`ng-maxlength`、`ng-required` 等。在上例的模板中，我们在每个 `input` 下，通过 `myForm` 的多个属性来判断对应的错误是否要显示。

我们继续来看表单元素中 `ng-model` 的用法。该指令专门用于表单元素，它可以创建数据的双向绑定。数据绑定相关的内容已经在前面讨论过。当用户输入的内容发生改变时，通过 `ng-model` 绑定的 `$scope` 属性也会被更新；同时，借助双向绑定，反向的过程也是有效的。我们在 `controller` 方法中修改一个数据，与之绑定的表单标签也会更新。在 Angular 中，我们更推荐用 `ng-model` 获取表单元素的取值，而不是通过 `name` 属性。这是因为在 Angular 中，表单元素的 `name` 属性主要用于数据验证。

图 8-6、图 8-7 以及图 8-8 展示了用户在浏览器中看到的最终效果。

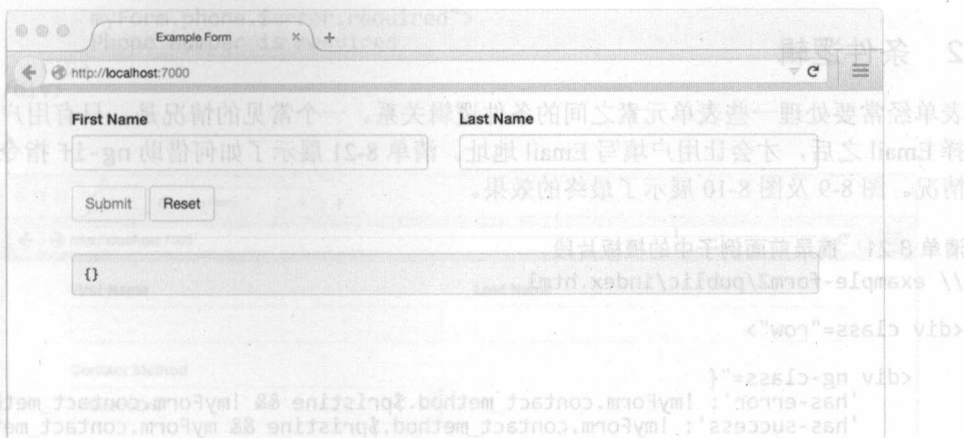


图 8-6 我们的表单最初的状态。这个例子中，我们可以实时地看到`$scope.model`的变化

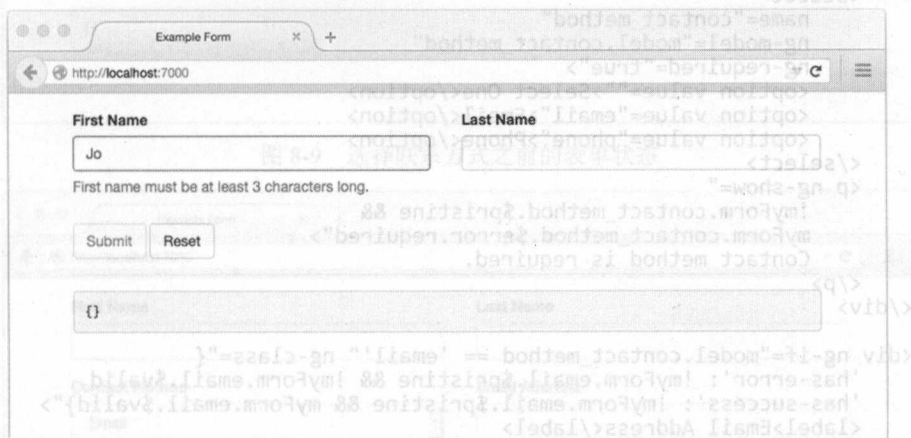


图 8-7 当用户输入信息时，表单会自动做出反馈。这里我们告诉用户“First Name”字段必须至少包含三个字符

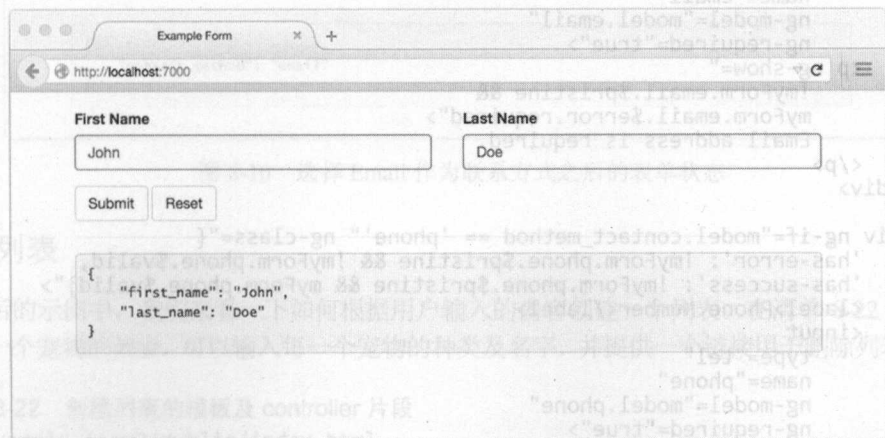


图 8-8 当用户正确输入所有内容之后，表单的状态

8.7.2 条件逻辑

表单经常要处理一些表单元素之间的条件逻辑关系。一个常见的情况是，只有用户在联系方式中选择 Email 之后，才会让用户填写 Email 地址。清单 8-21 展示了如何借助 `ng-if` 指令实现上面提到的情况。图 8-9 及图 8-10 展示了最终的效果。

清单 8-21 摘录前面例子中的模板片段

// example-form2/public/index.html

```
<div class="row">
  <div ng-class="{
    'has-error': !myForm.contact_method.$pristine && !myForm.contact_method.$valid,
    'has-success': !myForm.contact_method.$pristine && myForm.contact_method.$valid
  }">
    <label>Contact Method</label>
    <select
      name="contact_method"
      ng-model="model.contact_method"
      ng-required="true">
      <option value="">Select One</option>
      <option value="email">Email</option>
      <option value="phone">Phone</option>
    </select>
    <p ng-show="
      !myForm.contact_method.$pristine &&
      myForm.contact_method.$error.required">
      Contact method is required.
    </p>
  </div>

  <div ng-if="model.contact_method == 'email'" ng-class="{
    'has-error': !myForm.email.$pristine && !myForm.email.$valid,
    'has-success': !myForm.email.$pristine && myForm.email.$valid}">
    <label>Email Address</label>
    <input
      type="email"
      name="email"
      ng-model="model.email"
      ng-required="true">
    <p ng-show="
      !myForm.email.$pristine &&
      myForm.email.$error.required">
      Email address is required.
    </p>
  </div>

  <div ng-if="model.contact_method == 'phone'" ng-class="{
    'has-error': !myForm.phone.$pristine && !myForm.phone.$valid,
    'has-success': !myForm.phone.$pristine && myForm.phone.$valid}">
    <label>Phone Number</label>
    <input
      type="tel"
      name="phone"
      ng-model="model.phone"
      ng-required="true">
    <p ng-show="
      !myForm.phone.$pristine &&
```

```

myForm.phone.$error.required">
Phone number is required.
</p>
</div>
</div>

```

图 8-9 选择联系方式之前的表单状态

图 8-10 选择 Email 作为联系方式之后的表单状态

8.7.3 列表

在最后的示例中，我们来看一下如何根据用户输入的内容创建一个列表。在清单 8-22 中，表单帮助用户创建一个宠物的列表，可以输入每一个宠物的种类及名字，并提供一个链接用于删除列表中的某一项。

清单 8-22 创建列表的模板及 controller 片段

```
// example-form3/public/index.html
```

```
<div class="row">
```

```

<div>
  <h2>Pets</h2> <small><a ng-click="addPet()">Add Pet</a></small>
</div>
</div>

<div class="row" ng-repeat="pet in model.pets">
  <div>
    <label>Pet Type</label>
    <select
      ng-attr-name="pet_type{{$index}}"
      ng-model="pet.type"
      required>
      <option value="">Select One</option>
      <option value="cat">Cat</option>
      <option value="dog">Dog</option>
      <option value="Goldfish">Goldfish</option>
    </select>
  </div>

  <div ng-class="{
    'has-error': !myForm.last_name.$pristine && !myForm.last_name.$valid,
    'has-success': !myForm.last_name.$pristine && myForm.last_name.$valid
  }">
    <label>
      Pet's Name <small class="pull-right">
        <a ng-click="removePet(pet)">Remove Pet</a></small>
      </label>
    <input
      type="text"
      ng-attr-name="pet_name{{$index}}"
      ng-model="pet.name"
      ng-minlength="3"
      ng-maxlength="15"
      required>
    <p ng-show="
      !myForm.last_name.$pristine &&
      myForm.last_name.$error.required">
      Last name is required.
    </p>
    <p ng-show="
      !myForm.last_name.$pristine &&
      myForm.last_name.$error.minlength">
      Last name must be at least 3 characters long.
    </p>
    <p ng-show="
      !myForm.last_name.$pristine &&
      myForm.last_name.$error.maxlength">
      Last name can have no more than 15 characters.
    </p>
  </div>
</div>
</div>
// example-form5/public/app/index.js

$scope.addPet = function() {
  $scope.model.pets.push({});
};

$scope.removePet = function(pet) {
  $scope.model.pets.splice($scope.model.pets.indexOf(pet), 1);
};

```


在清单 8-22 中，我们使用 `ng-repeat` 指令遍历了 `$scope.model.pets` 数组的每一项。注意，在 `ng-repeat` 片段中，我们可以用 `{{ $index }}` 获取当前行在数组中的位置。借助这个信息，我们可以为表单提供唯一的名字，以便于数据验证。

在模板中，我们在列表的顶端提供了一个 Add Pet 链接。单击这个链接，会触发我们在 controller 中定义的 `addPet()` 方法。在这个方法中，我们为 `$scope.model.pets` 添加了一个空对象。同时，我们也提供了一个链接用于移除。当单击这个链接时，当前条目绑定的对象会被传入 `removePet()` 方法，这条数据也就从数组中被移除。

图 8-11 展示了最终的效果。

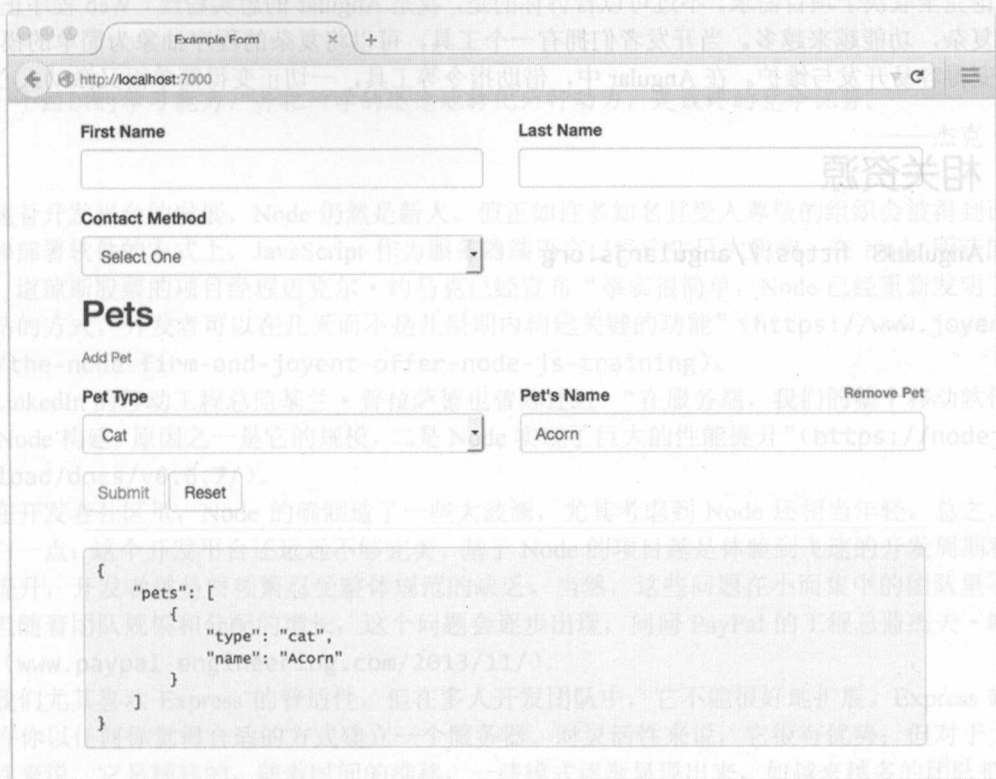


图 8-11 最后一个示例的最终效果

8.8 小结

在本章的开始，我们对比了传统的“命令式编程”与 Angular 中用到的“声明式编程”。尽管这两种方式各有利弊，但是就表单开发来说，Angular 的方式更好，这一点毋庸置疑。

Angular 正在逐渐地进化为一个可以支持大规模程序的框架，但这已经背离了它的初衷。Angular 的联合创始人 Miško Hevery 承认，Angular 最初只是为了改善 Web 表单的开发。了解这一点非常重要，因为这也说明了 Angular 适合的项目类型。

自发布以来, Angular 吸引了大量关注, 当然, 这其中大多数人都持正面态度。该框架中提到的指令、依赖注入等概念对客户端开发产生了较大的影响, 并引发了开发者对于相似的框架需要提供哪些内容的讨论。

与此同时, 持反对意见的开发者也越来越多。大多数批评的声音来自于对 Angular 性能的担忧。Angular 中双向绑定采用的“脏检查”确实并不高效。不过, 就作者的经验来说, Angular 的性能足以应付多数的场景。在本章的写作过程中, Angular 的一个新版本 (v2.0) 正在紧锣密鼓的筹备当中。该版本几乎对整个框架进行了重写, 届时这些问题中相当大的一部分 (不是所有) 将会得到解决。

如果你想知道 Angular 是否适合自己的项目, 那么并不存在“是”或“不是”这样简单的答案。最终结论完全取决于项目需求。不过可以告诉你的是, 我是 Angular 的忠实粉丝! Web 程序正在变得越来越复杂, 功能越来越多。当开发者们拥有一个工具, 可以将复杂的程序抽象为简单的模式, 程序自然变得容易开发与维护。在 Angular 中, 借助指令等工具, 一切正变得简单而又激动人心。

8.9 相关资源

- AngularJS: <https://angularjs.org>

Kraken

一个组织的学习能力，并把所学的迅速地转化为行动力，是最终的竞争优势。

——杰克·韦尔奇

随着开发平台的发展，Node 仍然是新人。但正如许多知名且受人尊敬的组织会被得到证明，在开发和部署软件的方式上，JavaScript 作为服务器端语言已经产生巨大影响。在 Node 所获的多项殊荣中，道琼斯股票的项目经理迈克尔·约马克已经宣布“事实很简单，Node 已经重新发明了我們创建网站的方式。开发者可以在几天而不是几星期内构建关键的功能”(<https://www.joyent.com/blog/the-node-firm-and-joyent-offer-node-js-training>)。

LinkedIn 的移动工程总监基兰·普拉萨德也曾陈述过：“在服务端，我们的整个移动软件栈都完全由 Node 构建。原因之一是它的规模，二是 Node 实现了巨大的性能提升”(<https://nodejs.org/download/docs/v0.6.7/>)。

在开发者社区里，Node 的确制造了一些大波澜，尤其考虑到 Node 还相当年轻。总之，让我们先明白一点：这个开发平台还远远不够完美。基于 Node 的项目越是体验到飞速的开发周期和骄人的性能提升，开发者越是要频繁忍受整体规范的缺乏。当然，这些问题在小而集中的团队里不会很明显，但随着团队规模和分配的增长，这个问题会逐步出现，问问 PayPal 的工程总监杰夫·哈瑞尔就好了 (www.paypal-engineering.com/2013/11/)。

我们尤其喜欢 Express 的普适性，但在多人开发团队中，它不能很好地扩展。Express 缺乏规范并允许你以任何你觉得合适的方式建立一个服务器。对灵活性来说，它很有优势；但对于大团队的稳定性来说，它是糟糕的。随着时间的推移，一些模式逐渐显现出来，如越来越多的团队把 Node.js 替换为 Kraken.js；Kraken.js 本质上并不是一个框架，但作为 Express 上层的规范，它能够更好地适应更大的开发团队。我们希望工程师们能聚焦于构建应用，无需聚焦于搭建环境。

本章将介绍 Kraken。它由 PayPal 工程师开发，为基于 Express 的应用提供了一层安全可伸缩层。本章包含以下几个主题：

- 环境感知的配置
- 基于配置的中间件注册信息
- 注册模式化路由
- Dust 模板引擎
- 国际化和本地化
- 增强安全技术

■ **注意** Kraken 构建于已经存在的 Express 基础之上，在这个分类下，Express 极简 Web 框架的 API 已经变成了事实标准。因此，本章假设读者已经有了 Express 的基础。本章的部分内容也会提及本书中 Grunt、Yeoman、Knex 和 Bookshelf 章节中的概念。如果你对这些问题不熟，在继续阅读本章之前可以先学习这些章节。

9.1 环境感知的配置

当应用开发、测试、预发布和发布时，它们自然地随着相应的环境演进，每一步需要各自唯一的配置规则。例如，思考一下图 9-1，它阐述了一系列应用从整合到发布部署管线的过程。

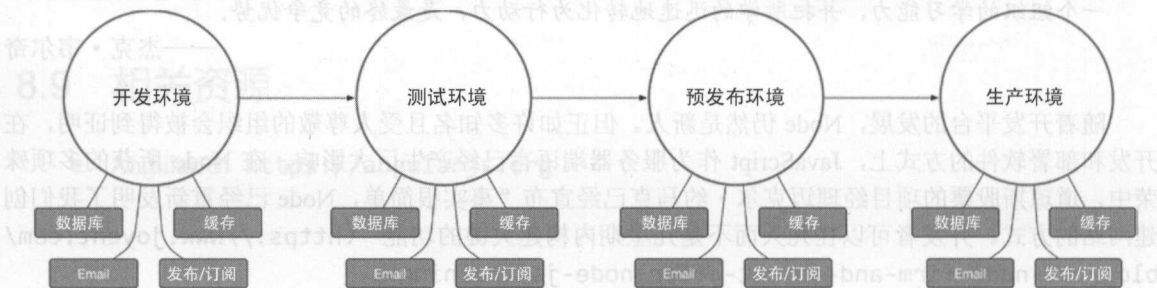


图 9-1 基于应用的环境所需的唯一配置

正如图 9-1 中在各个环境下逐步发展的应用，其设置说明了如何与不同的外部服务相连接，在不同外部服务上依赖必须相应改变。Kraken 的 `confit` 库为开发者提供了标准规范，用于实现为 Node 应用提供简单、环境感知的配置层这一目标。

Confit 通过加载默认 JSON 文件（通常该文件命名为 `config.json`）运行。然后，Confit 试图通过 `NODE_ENV` 环境变量加载额外配置文件。

如果找到某个基于环境的配置文件，它指定的任何配置都会和定义在默认配置中的定义递归合并。Express 作为极简的 Node 框架，Kraken 构建在它已经稳固的体系之上。本章的项目 `confit-simple` 提供了简单的应用，它依赖于 `confit` 来决定它的配置。清单 9-1 展示了 `confit` 初始化的过程，清单 9-2 则展示了项目/`config` 文件夹的内容，其内容引导 `confit` 搜索配置文件。

清单 9-1 初始化配置

```
// confit-simple/index.js
var confit = require('confit');
var prettyjson = require('prettyjson');
var path = require('path');
var basedir = path.join(__dirname, 'config');

confit(basedir).create(function(err, config) {
  if (err) {
    console.log(err);
    process.exit();
  }
  console.log(prettyjson.render({
    'email': config.get('email'),
  }));
});
```



```

    'cache': config.get('cache'),
    'database': config.get('database')
  }));
});

```

清单 9-2 /config 文件夹下的内容

```

// 默认配置
// config-simple/config/config.json
{
  // SMTP server settings
  "email": {
    "hostname": "email.mydomain.com",
    "username": "user",
    "password": "pass",
    "from": "My Application <noreply@myapp.com>"
  },
  "cache": {
    "redis": {
      "hostname": "cache.mydomain.com",
      "password": "redis"
    }
  }
}

// 开发环境配置
// config-simple/config/development.json
{
  "database": {
    "postgresql": {
      "hostname": "localhost",
      "username": "postgres",
      "password": "postgres",
      "database": "myapp"
    }
  },
  "cache": {
    "redis": {
      "hostname": "localhost",
      "password": "redis"
    }
  }
}

// 生产环境配置
// config-simple/config/production.json
{
  "database": {
    "postgresql": {
      "hostname": "db.myapp.com",
      "username": "postgres",
      "password": "super-secret-password",
      "database": "myapp"
    }
  },
  "cache": {
    "redis": {
      "hostname": "redis.myapp.com",
      "password": "redis"
    }
  }
}

```

在继续讨论接下来的内容之前，注意我们项目的默认配置文件的 email 属性，它提供了对 E-mail 服务器的链接设置，然而其他任何基于环境的配置文件内都没有指定该属性。相比之下，默认配置还提供了对 Redis 缓存服务器的链接配置，其配置在内嵌设置 cache 下面，然而，每个基于环境的配置都覆盖了这个属性的信息。

还要注意一点，默认配置在 email 属性上包含了一个注释。注释不作为 JSON 标准的一部分。一般情况下，如果我们试图用 Node 的 `require()` 方法引用该文件，在解析该文件时会抛出一个错误。然而，Confit 会在 Node 尝试解析该文件之前剔除该注释，这项功能可以让我们在配置文件中根据需要来添加注释。

当项目使用开发模式下设置的 `NODE_ENV` 环境变量运行后，清单 9-3 显示了打印到控制台的输出日志。

清单 9-3 在开发模式下运行 `confit-simple` 项目

```
$ export NODE_ENV=development && node index
```

```
email:
  hostname: email.mydomain.com
  username: user
  password: pass
  from:      My Application <noreply@myapp.com>
cache:
  redis:
    hostname: localhost
    password: redis
database:
  postgresql:
    hostname: localhost
    username: postgres
    password: postgres
    database: myapp
```

■ **注意** 在清单 9-3 中，终端运行的 `$ export NODE_ENV=development` 命令用于设置 `NODE_ENV` 环境变量。该命令仅适用于 UNIX 和类似 UNIX 的系统（包括 OS X）。而 Windows 用户需要运行 `$ set NODE_ENV=development`。记住这一点很重要，即如果 `NODE_ENV` 环境变量没有设置，`confit` 会假设应用运行在开发环境下。

正如清单 9-3 所示，`confit` 通过合并开发环境下的 `config/development.json` 和默认配置 `config/config.json` 来编译项目配置对象，其中指定在 `development.json` 中的配置优先级更高。最终结果是，项目对象继承了仅仅配置在默认文件 `config.json` 中的 email 属性，连同开发环境下的缓存和数据库设置。在清单 9-1 中，可以通过配置对象的 `get()` 方法获取设置。

■ **注意** 除了获取顶级配置（如清单 9-1 所示的数据库），配置对象的 `get()` 方法还能用于获取使用：作为分隔符的内嵌配置。例如，我们可以通过 `config.get('database:postgresql')` 来引用项目的 `postgresql` 设置。

清单 9-4 中再次运行了 `confit-simple` 项目，只是这次我们设置的 `NODE_ENV` 环境变量用于生产环境。不出所料，输出结果显示配置对象继承了 `config.json` 中的 email 属性，也显示了继承自 `production.json` 中的缓存和数据库属性。

清单 9-4 在生产模式下运行 confit-simple 项目
\$ export NODE_ENV=production && node index

```
email:
  hostname: email.mydomain.com
  username: user
  password: pass
  from: My Application <noreply@myapp.com>
cache:
  redis:
    hostname: redis.myapp.com
    password: redis
database:
  postgresql:
    hostname: db.myapp.com
    username: postgres
    password: super-secret-password
    database: myapp
```

Shortstop 处理器

如前面例子所示，Confit 设计的目的是用于处理 JSON 配置文件。作为一种配置文件格式，JSON 很易于处理，但由于其灵活性，它偶尔会有一点不足之处。Confit 通过支持作为“Shortstop 处理器”的插件弥补了这些缺点。通过清单 9-5 中的例子，该示例中 confit 的核心库中引用了两个 Shortstop 处理器，分别是 import 和 config 处理器。

清单 9-5 import 和 config 处理器使用方式示例

```
// confit-shortstop/config/config.json
{
  // 'import' 处理器让我们可以把某个属性的值设为指定的 JSON 配置文件。
  "app": "import:./app",
  // 'config' 处理器让我们可以把某个属性的值设为某个属性的引用，
  // 注意本例中使用 '.' 字符作为分隔符。
  "something_else": "config:app.base_url"
}

// confit-shortstop/config/app.json
{
  // 示例标题
  "title": "My Demo Application",
  // Web 客户端可以访问的 base URL
  "base_url": "https://myapp.com",
  // API 可以访问的 base URL
  "base_api_url": "https://api.myapp.com"
}
```

清单 9-6 显示了本章示例程序 confit-shortstop 运行时打印到控制台的输出。在这个示例中，导入 Shortstop 处理器使我们能够通过一个单独的 JSON 文件计算 app 的属性。config 处理器允许我们引用其他小节中已经存在的值来设置配置项。

清单 9-6 本章 confit-shortstop 项目的输出
\$ node index.js

```
app:
  title: My Demo Application
```

```
base_url:      https://myapp.com
base_api_url: https://api.myapp.com
something_else: https://myapp.com
```

尽管 `confit` 本身只包含对刚刚提及的两个 `Shortstop` 处理器 (`import` 和 `config`) 有支持, 但一些额外的处理器也很实用, 可以在 `shortstop-handlers` 模块中找到。

让我们看四个示例。清单 9-7 展示了 `confit-shortstop-extras` 项目示例。该脚本除了少数的不同, 很大程度上和清单 9-1 中的示例相同。这个示例从 `shortstop-handlers` 模块中引入额外的处理器。同样, 不同于传递项目配置文件夹路径来初始化 `confit(basedir)`, 我们传递了 `options` 对象。这个对象为 `basedir` 指定一个值, 同时还传入一个 `protocols` 对象, 为 `confit` 传递了我们后续可能会使用的额外 `Shortstop` 处理器引用。

清单 9-7 项目 `confit-shortstop-extras` 中的 `index.js` 脚本

```
// confit-shortstop-extras/index.js
```

```
var confit = require('confit');
var handlers = require('shortstop-handlers');
var path = require('path');
var basedir = path.join(__dirname, 'config');
var prettyjson = require('prettyjson');

confit({
  'basedir': basedir,
  'protocols': {
    // 'file' 处理器让我们能为属性的值设置一个额外的 (非 JSON) 文件。
    // 默认情况下, 该文件会以 Buffer 对象加载。
    'file': handlers.file(basedir /* Folder from which paths should be resolved */, {
      'encoding': 'utf8' // Convert Buffers to UTF-8 strings
    }),
    // 'require' 处理器让我们能把属性的值设置为某个导出的模块。
    'require': handlers.require(basedir),
    // 'glob' 处理器让我们能把属性的值设置为一个匹配指定模式的文件数组。
    'glob': handlers.glob(basedir),
    // 'path' 处理器支持解析相对路径。
    'path': handlers.path(basedir)
  }
}).create(function(err, config) {
  if (err) {
    console.log(err);
    process.exit();
  }
  console.log(prettyjson.render({
    'app': config.get('app'),
    'something_else': config.get('something_else'),
    'ssl': config.get('ssl'),
    'email': config.get('email'),
    'images': config.get('images')
  }));
});
```

本例使用了四个额外的 `Shortstop` 处理器 (从 `shortstop-handlers` 模块中导入)。

- `file`: 为属性的值设置一个额外的 (非 JSON) 文件
- `require`: 把属性的值设置为某个 Node 导出的模块 (对仅能在运行环境中确定的动态值很有用)
- `glob`: 把属性的值设置为一个匹配指定模式的文件数组
- `path`: 把属性的值设置为相对于某个文件的路径

清单 9-8 显示了 `confit-shortstop-extras` 项目的默认配置文件。清单 9-9 显示了项目启动后打印到

控制台的输出信息。

清单 9-8 confit-shortstop-extras 项目默认配置文件

// confit-shortstop-extras/config/config.json

```
{
  "app": "import:./app",
  "something_else": "config:app.base_url",
  "ssl": {
    "certificate": "file:./certificates/server.crt",
    "certificate_path": "path:./certificates/server.crt"
  },
  "email": "require:./email",
  "images": "glob:.../public/images/**/*.jpg"
}
```

清单 9-9 confit-shortstop-extras 项目输出

\$ export NODE_ENV=development && node index

```
app:
  title:      My Demo Application
  base_url:   https://myapp.com
  base_api_url: https://api.myapp.com
  something_else: https://myapp.com
ssl:
  certificate_path: /opt/confit-shortstop-extras/config/certificates/server.crt
  certificate:
    ""
```

-----BEGIN CERTIFICATE-----

```
MIIDNjCCAAoYCCQDy8G1RKCEz4jANBgkqhkiG9w0BAQUFAADCBLMAKGA1UEBhMC
VVMxEjAQBgNVBAgTCVRlbn5lc3NlZTESMBAGA1UEBxMjTmFzaHJpZGxIMSEwHwYD
VQKEXhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQxFDASBgNVBAMUCyoubXlhcHAu
Y29tMSAwHgYJKoZIhvcNAQkBFhFzdXBwb3J0QG15YXBwLmNvbTAeFw0xNTA0MTkw
MDA4MzRaFw0xNjA0MTgwMDA4MzRaMIGQMqswCQYDVQQGEwJVUzESMBAGA1UECBMj
VGvubmVzc2VlMRlWIAEYDVQQHEw10YXNodm1sbGUxITAFBgNVBAAoTGEudGVybWV0
IFdpZGdpdHMGUHR5IEh0ZDEUMBIGA1UEAxQLKi5teWwFwC5jb20xIDAeBgkqhkiG
9w0BCQEWEXN1cHBvcnRABXlhcHAuY29tMIIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A
MIIBCGKCAQEAyBfXmVlMjP7VCU5w70okfJX/oEYtrQI1Z0AXnErryQQWwZpH0lu
ZHuZ8sB3mMBH3jjU+rxr4C2dFlXxWDRp8nYt+qfd1aiBKjYxMda2QMwXviT0Td9b
kPFBCaPQpMrzexwTwK/edoaxzqs/IxMs+n1PfvpwU0uPk6UbwFwWc8UQSWrmbGJw
UEfs1X9K0Svt85IdrdQ1hQP2fBhHvt/xVVPf11ZW1yBrWscVHBOJ04RyZSGclayg
7LP+VHMvkVNm0au/cmCWThHtRt3aXhxAZtgkI9IT2G4B9R+7ni8eXw5TL165bhr1
Gt7fMK2HnXclPtD3+vy9EnM+XqYXahXFGWIDAQABMA0GCSqGSIb3DQEBBQUAA4IB
AQDHH+QmuWk0Bx1kqUoL1Qxtqgf7s81eKoW5X3Tr4ePFXQbwmCZKHEudC98XckI2j
qGA/SViBr+nb0f6pTnBhAoYV0IQd4YT3qv0+m3otGQ7NQk02HwD30UG9khHe2mG
k8Z7pF0puw3lbtGkadi3JSsS1fJGs9hy2vSzRulgoZozT3HJ+2Sjpiwy7QAR0aF
jqMC+HcP38zZkTWj1s045HRCU1HdPjr0U3oJtupiU+HAmNpf+vdQnxS6aM5nzc7G
tZq74ketSxEXYTU8gjfM1R4gBewfPmu2KGuHNV51GAjWgm9wLFPFvMMYjciEPB3k
Mla9+pYx1YvXiyJmOnUwsaop
```

-----END CERTIFICATE-----

""

email:

```
hostname: smtp.myapp.com
username: user
password: pass
from:      My Application <noreply@myapp.com>
```

images:

```
- /opt/confit-shortstop-extras/public/images/cat1.jpg
- /opt/confit-shortstop-extras/public/images/cat2.jpg
- /opt/confit-shortstop-extras/public/images/cat3.jpg
```

9.2 注册基于配置的中间件

Express 处理 HTTP 到达请求的方式是将其推入一系列配置“中间件”的函数中，如图 9-2 所示。

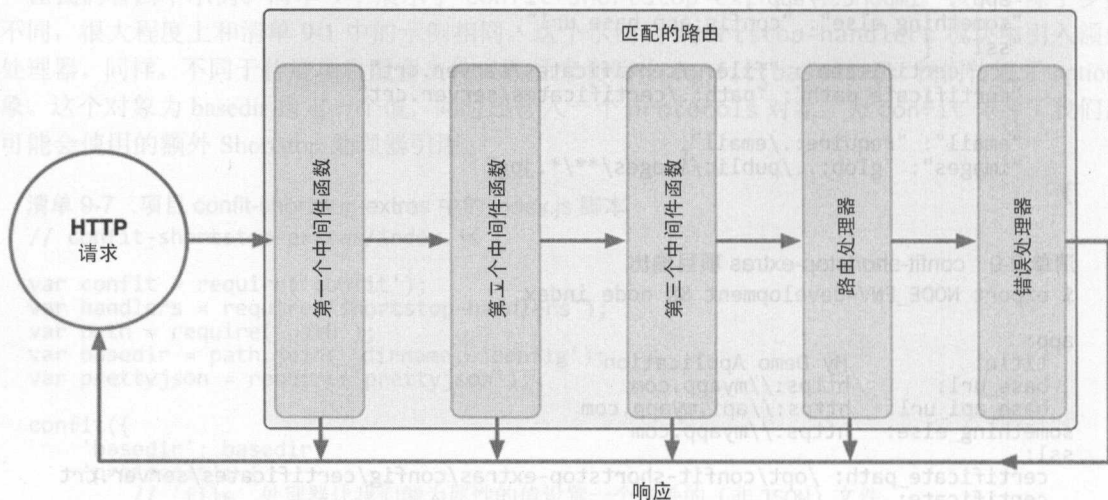


图 9-2 Express 中间件调用序列

处理过程的每一步，有效中间件函数都能够：

- 修改 HTTP 请求对象
- 修改 HTTP 响应对象
- 执行额外代码
- 关闭请求-响应循环
- 调用序列中下一个中间件函数

按照清单 9-10 所示例子中方式，思考一下，其显示了一个简单的 Express 应用，该应用依赖三个中间件模块：`morgan`、`cookie-parser` 和 `ratelimit-middleware`。当应用处理 HTTP 请求时，触发如下四个步骤。

1. `morgan` 模块将请求打印至控制台。
2. `cookie-parser` 模块解析请求头的 `Cookie` 模块并将值赋给请求对象的 `cookies` 属性中。
3. `ratelimit-middleware` 模块对试图频繁访问应用的客户端做限制。
4. 最终调用合适的路由处理器。

清单 9-10 依赖了三个中间件模块的 Express 应用

```
// middleware1/index.js
```

```
var express = require('express');
// 打印请求日志
var morgan = require('morgan');
// 使用从请求头中解析得到的 'Cookie' 填充 'req.cookies' 属性
var cookieParser = require('cookie-parser');
// 配置 API 访问率限制
```

```

var rateLimit = require('ratelimit-middleware');

var app = express();
app.use(morgan('combined'));
app.use(cookieParser());
app.use(rateLimit({
  'burst': 10,
  'rate': 0.5,
  'ip': true
}));

app.get('/animals', function(req, res, next) {
  res.send(['squirrels', 'aardvarks', 'zebras', 'emus']);
});

app.listen(7000);

```

这种方式为开发者提供了相当大的灵活度，使开发者能在请求-响应循环的任意一点中执行特定逻辑。通过把执行不必要的任务的职责代理给第三方中间件模块，Express 只需要维护相对小的封装。但这种方法的灵活性同样引入了问题，即如何管理应用及开发该应用的团队在规模和复杂度增长时面临的问题。

Kraken 的 meddleware 模块简化了中间件的管理，方式是给 Express 应用提供基于配置的中间件注册过程。这样做能让开发者对 Express 应用的中间件提供标准方法，包括该依赖哪个模块、该以何种顺序加载、该为每个模块传入什么选项。清单 9-11 展示了上一个示例的更新版本，其中的 meddleware 模块管理了所有中间件函数的注册。

清单 9-11 使用 meddleware 模块实现基于配置的中间件注册

```

// middleware2/index.js

var express = require('express');
var confit = require('confit');
var meddleware = require('meddleware');
var app = express();
var path = require('path');
confit(path.join(__dirname, 'config')).create(function(err, config) {
  app.use(meddleware(config.get('middleware')));
  app.get('/animals', function(req, res, next) {
    res.send(['squirrels', 'aardvarks', 'zebras', 'emus']);
  });
  app.listen(7000);
});

// middleware2/config/config.json
{
  "middleware": {
    "morgan": {
      // 中间件模块开启/关闭开关
      "enabled": true,
      // 指定中间件模块注册顺序
      "priority": 10,
      "module": {
        // 已安装模块名 (或模块文件路径)
        "name": "morgan",
        // 用于传递给模块工厂函数的参数
        "arguments": ["combined"]
      }
    }
  }
}

```

```

    },
    "cookieParser": {
      "enabled": true,
      "priority": 20,
      "module": {
        "name": "cookie-parser"
      }
    },
    "rateLimit": {
      "enabled": true,
      "priority": 30,
      "module": {
        "name": "ratelimit-middleware",
        "arguments": [{
          "burst": 10,
          "rate": 0.5,
          "ip": true
        }]
      }
    }
  }
}

```

在 Kraken 的 meddleware 模块的帮助下, 各种第三方中间件模块的管理由应用代码中移动到标准配置文件中。如此一来, 应用不仅更有条例, 也更易理解和修改。

事件通知

通过 meddleware 模块, 中间件模块注册在 Express 上, 应用也会发出相应的事件, 让开发者很容易知道哪个中间件函数被加载以及按什么顺序加载 (见清单 9-12)。

清单 9-12 中间件通过 meddleware 模块注册时发出事件

```

var express = require('express');
var confit = require('confit');
var meddleware = require('meddleware');
var app = express();
var path = require('path');

confit(path.join(__dirname, 'config')).create(function(err, config) {
  // 监听所有中间件注册
  app.on('middleware:before', function(data) {
    console.log('Registering middleware: %s', data.config.name);
  });

  // 监听特定中间件注册
  app.on('middleware:before:cookieParser', function(data) {
    console.log('Registering middleware: %s', data.config.name);
  });

  app.on('middleware:after', function(data) {
    console.log('Registered middleware: %s', data.config.name);
  });

  app.on('middleware:after:cookieParser', function(data) {
    console.log('Registered middleware: %s', data.config.name);
  });

  app.use(meddleware(config.get('middleware')));
}

```



```

app.get('/animals', function(req, res, next) {
  res.send(['squirrels', 'aardvarks', 'zebras', 'emus']);
});

app.listen(7000);
});

```

9.3 结构化路由注册

前一节学习了 Kraken 的 `meddleware` 模块可以简化中间件函数注册,其方式是把加载和配置中间件函数的逻辑移动到标准 JSON 中。以几乎相同的方式,对于路由结构缺失的 Express, Kraken 的 `enrouten` 模块使用相同理念为其提供了结构。

在简单的 Express 应用中,少量路由定义可以放在一个单独的模块中。然而,由于应用的深度和复杂度会逐渐增长,这样的组织结构(或者说缺少结构)会使应用很快变得难以处理。`enrouten` 模块通过提供三种方法解决这个问题,使 Express 路由可以用一致的、结构化的方式定义。

9.3.1 索引配置

使用 `enrouten` 的索引配置选项,可以指定简单模块的路径。这个路径随后会被加载,并传递一个已经挂载到根路径上的 Express 路由实例。这个选项为开发者提供了最简单的路由定义方式,同时它不会强制使用任何特定的组织结构。虽然这个选项为新的应用提供了一个良好的开端,但也必须注意不要滥用。该选项经常和 `enrouten` 的 `routes` 和 `config` 配置选项结合在一起。稍后即将介绍路由配置选项。

清单 9-13 展示了一个简单的 Express 应用,它使用了 `confit`、`meddleware` 和 `enrouten` 的多个配置。清单 9-14 显示了传递到 `enrouten` 索引选项的模块内容。本节中后续例子会在这个示例上继续构建。

清单 9-13 使用 `confit`、`meddleware` 和 `enrouten` 配置 Express 应用

```

// enrouten-index/index.js

var express = require('express');
var confit = require('confit');
var handlers = require('shortstop-handlers');
var meddleware = require('meddleware');
var path = require('path');
var configDir = path.join(__dirname, 'config');
var app = express();

confit({
  'basedir': configDir,
  'protocols': {
    'path': handlers.path(configDir),
    'require': handlers.require(configDir)
  }
}).create(function(err, config) {
  app.use(meddleware(config.get('middleware')));
  app.listen(7000);
  console.log('App is available at: http://localhost:7000');
});

// enrouten-index/config/config.json

```

```
{
  "middleware": {
    "morgan": {
      "enabled": true,
      "priority": 10,
      "module": {
        "name": "morgan",
        "arguments": ["combined"]
      }
    },
    "enrouten": {
      "enabled": true,
      "priority": 30,
      "module": {
        "name": "express-enrouten",
        "arguments": [
          {
            "index": "path:../routes/index"
          }
        ]
      }
    }
  }
}
```

清单 9-14 传递给 enrouten 索引选项的模块内容

// enrouten-index/routes/index.js

```
module.exports = function(router) {
  router.route('/')
    .get(function(req, res, next) {
      res.send('Hello, world.');
```

```
});
```

```
router.route('/api/v1/colors')
  .get(function(req, res, next) {
    res.send([
      'blue', 'green', 'red', 'orange', 'white'
    ]);
```

```
});
```

```
};
```

9.3.2 目录配置

清单 9-15 展示了 enrouten 的目录配置选项。当设置了该选项，enrouten 会递归扫描指定文件夹下的内容，寻找那些导出函数接受一个单独参数的模块。对每个找到的模块，enrouten 会为其传递一个 Express 路由实例，该实例已经挂载到模块的目录结构路径下——这是一种“规范配置”方法。

清单 9-15 设置 enrouten 的目录配置选项

// enrouten-directory/config/config.json

```
{
  "middleware": {
    "enrouten": {
      "enabled": true,
      "priority": 10,
      "module": {
```

```

    "name": "express-enrouten",
    "arguments": [{ "directory": "path:../routes" }]
  }
}
}
}

```

图 9-3 显示了项目的/routes 文件夹结构,同时清单 9-16 显示了/routes/api/v1/accounts/index.js 模块的内容。根据模块在/routes 文件夹下的位置,每个路由 URL 都会加上一个/api/v1/accounts 前缀。

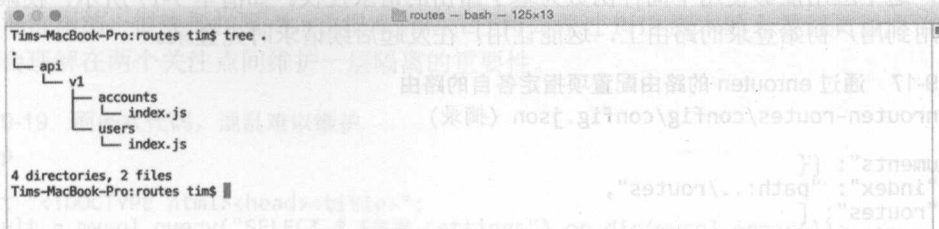


图 9-3 项目的/routes 文件夹结构

清单 9-16 /api/v1/accounts 控制器

```
// enrouten-directory/routes/api/v1/accounts/index.js
```

```

var _ = require('lodash');
var path = require('path');

module.exports = function(router) {

  var accounts = require(path.join(APPROOT, 'models', 'accounts'));

  /**
   * @route /api/v1/accounts
   */
  router.route('/')
    .get(function(req, res, next) {
      res.send(accounts);
    });

  /**
   * @route /api/v1/accounts/:account_id
   */
  router.route('/:account_id')
    .get(function(req, res, next) {
      var account = _.findWhere(accounts, {
        'id': parseInt(req.params.account_id, 10)
      });
      if (!account) return next(new Error('Account not found'));
      res.send(account);
    });
};

```

9.3.3 路由配置

enrouten 的目录配置项通过基于指定文件夹的结构自动决定应用的 API 结构,这项功能很适合那类赞成“约定优于配置”的开发者。这种方式以一种简单快速的方法,给 Express 路由提供了一个有

组织的一致结构。然而更复杂的应用最终还是会发现这种办法依然稍显局限。

那些有着复杂、深层次嵌套的路由 API 应用更容易发现 `enrouten` 的 `routes` 配置项的优点，它让开发者能够为应用的每个路由创建独立复杂的模块。API 端点、方法、处理器及特定于路由的中间件会在配置文件中指定——它以稍显烦琐的配置方式，为开发者提供很大程度的灵活性。

清单 9-17 显示了本章 `enrouten-routes` 项目模块中的部分代码。我们向 `enrouten` 的配置路由中传入一个对象数组，路由的入口描述文件描述了可以用作 `Express` 的路由。注意，除了要指定路由、HTTP 方法和处理器之外，每个入口同样有选项可以指定针对某个路由的中间件数组。

因此，这个应用能够使用中间件函数为基于路由的请求授权。如清单 9-17 所示，权限中间件函数还没应用到用户初始登录的路由上，这能让用户在发起后续请求时先登录。

清单 9-17 通过 `enrouten` 的路由配置项指定各自的路由

// `enrouten-routes/config/config.json` (摘录)

```
"arguments": [{
  "index": "path:../routes",
  "routes": [
    {
      "path": "/api/v1/session",
      "method": "POST",
      "handler": "require:../routes/api/v1/session/create"
    },
    {
      "path": "/api/v1/session",
      "method": "DELETE",
      "handler": "require:../routes/api/v1/session/delete",
      "middleware": [
        "require:../middleware/auth"
      ]
    },
    {
      "path": "/api/v1/users",
      "method": "GET",
      "handler": "require:../routes/api/v1/users/list",
      "middleware": [
        "require:../middleware/auth"
      ]
    }
  ], ...
}]
```

清单 9-18 显示了负责处理应用路由 `/api/v1/usersGET` 请求的模块内容。该模块导出了一个单独的函数，其接受标准参数 `req`、`res` 以及下一个 `Express` 路由处理器签名。

清单 9-18 `/routes/api/v1/users/list` 路由处理器

```
var models = require('../../../../lib/models');

module.exports = function(req, res, next) {
  models.User.fetchAll()
    .then(function(users) {
      res.send(users);
    })
    .catch(next);
};
```


9.4 Dust 模板

许多流行的 JavaScript 模板引擎（如 Mustache 和 Handlebars）以“简化逻辑”作为自己的卖点——这个特质描述了模板能帮助开发者清楚地分离应用的业务逻辑和展示层关注点。当用户处理的界面发生了重大变化，如果处理适当，这种分离使得变化发生时后台所做的更改最少（反之亦然）。

所谓“无逻辑”模板引擎通过强制开发者使用一系列规则，避免出现经常书写的“面条式代码”。这种代码混乱纠结，很难抓住重点，也很难解耦。任何处理过 PHP 脚本的人看到类似清单 9-19 中的内容，都会理解在两个关注点间维护一层隔离的重要性。

清单 9-19 面条式代码，混乱难以维护

```
<?php
print "<!DOCTYPE html><head><title>";
$result = mysql_query("SELECT * FROM settings") or die(mysql_error());
print $result[0]["title"] . "</title></head><body><table>";
print "<thead><tr><th>First Name</th><th>Last Name</th></tr></thead><tbody>";
$users = mysql_query("SELECT * FROM users") or die(mysql_error());
while ($row = mysql_fetch_assoc($users)) {
    print "<tr><td>" . $row["first_name"] . "</td><td>" . $row["last_name"] . "</td></tr>";
}
print "</tbody></table></body></html>";
?>
```

通过禁止在应用的视图层书写逻辑，无逻辑模板引擎避免开发者书写面条式代码。这种模板通常能够在所提供的数据库负载中引用值、迭代数组以及基于简单的布尔逻辑控制特定内容的开关。

不幸的是，虽然它不期望这种方式，但这类相当粗暴的方法反而经常会带来它想避免的问题。尽管无逻辑的模板引擎如 Handlebars 避免在模板自身中使用逻辑，但它依然不能消除在一开始就需要逻辑的需求。为模板准备数据时一定在某处会需要逻辑，而且往往使用无逻辑的模板会导致展示相关的逻辑被迫放在业务逻辑层。

Dust 是 Kraken 提倡使用的模板引擎，通过一种“更少的逻辑”而不是严格的“无逻辑”来寻求解决这种问题的手段。它允许开发者在模板内以“辅助工具”的形式来使用更复杂的逻辑，而不是将其书写在业务层。

9.4.1 上下文及引用

当使用 Dust 模板时，两个主要的组件会发挥作用：模板本身以及一个（选项的）对象字面量——它包含任何要在模板内引用的数据。清单 9-20 通过指定 Dust 作为 Express 应用的模板引擎，展示了这一过程。注意本例中使用的 adaro 模块。adaro 模块作为 Dust 的简单包装，把 Express 引入 Dust 所需要的额外启动逻辑抽象出来。它还包括一些本章中会提到的便捷辅助方法。

清单 9-20 把 Dust 作为模板引擎的 Express 应用

```
// dust-simple/index.js
var express = require('express');
var adaro = require('adaro');
```

```
var app = express();

/**
 * 默认, Dust 会在应用加载时, 把模板的内容缓存起来。
 * 在生产环境下, 这通常是一种推荐的做法。
 * 在本章的示例中没有启用这个方法, 这样你能在不重启 Express 的情况下修改模板并看到结果。
 */
app.engine('dust', adaro.dust({
  'cache': false
}));

app.set('view engine', 'dust');
app.use('/', express.static('./public'));

var data = {
  'report_name': 'North American Countries',
  'languages': ['English', 'Spanish'],
  'misc': {
    'total_population': 565000000
  },
  'countries': [
    {
      'name': 'United States',
      'population': 319999999,
      'english': true,
      'capital': { 'name': 'Washington D.C.', 'population': 660000 }
    },
    {
      'name': 'Mexico',
      'population': 118000000,
      'english': false,
      'capital': { 'name': 'Mexico City', 'population': 9000000 }
    },
    {
      'name': 'Canada',
      'population': 35000000,
      'english': true,
      'capital': { 'name': 'Ottawa', 'population': 880000 }
    }
  ]
};

app.get('/', function(req, res, next) {
  res.render('main', data);
});

app.listen(8000);
```

清单 9-20 展示了一个包含北美国家数组（被 Dust 引用为一个“上下文”）的对象字面量，而后将该对象传入 Dust 模板。清单 9-21 展示了模板的内容。在这个模板里，通过把键包裹在一对大括号中，引用数据。嵌入属性还能通过使用点记法（`{misc.total_population}`）来引用。

清单 9-21 Dust 模板 main

```
// dust-simple/views/main.dust
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

```

<title>App</title>
<link href="/css/style.css" rel="stylesheet">
</head>
<body>
  {!! 使用这种格式书写 Dust 注释。数据的引用方式则通过用一对花括号包裹期望的键，如下所示!!}
  <h1>{report_name}</h1>
  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Population</th>
        <th>Speaks English</th>
        <th>Capital</th>
        <th>Population of Capital</th>
      </tr>
    </thead>
    <tbody>
      {!! 模板可以循环访问可迭代对象 !!}
      {#countries}
        <tr>
          <td>{name}</td>
          <td>{population}</td>
          <td>{?english}Yes{:else}No{/english}</td>
          {#capital}
            <td>{name}</td>
            <td>{population}</td>
          {/capital}
        </tr>
      {/countries}
    </tbody>
  </table>
  <h2>Languages</h2>
  <ul>
    {#languages}
    <li>{.}</li>
    {/languages}
  </ul>
  <h2>Total Population: {misc.total_population}</h2>
</body>
</html>

```

9.4.2 片段

在 Dust 处理渲染过程时，它在模板内通过应用一个或更多的“上下文”来引用数据。这个简单的模板只有一个上下文，用于引用传入的最外层 JSON 对象。例如，思考清单 9-21 中的模板，其中使用了两个引用：{report_name} 和 {misc.total_population}。Dust 处理这些引用的方式为从清单 9-20 所示的对象搜索匹配属性（从最外层开始）。

Dust 片段通过创建额外的上下文，提供了便捷的方法来让模板获取嵌入属性，而不需要引用最外层对象。例如，思考清单 9-22，其中创建了一个新的 {#misc}...{/misc} 上下文，这使得可以用简短的语法访问属性。

清单 9-22 创建一个新 Dust 片段

```

// Template
<h1>{report_name}</h1>
{#misc}
<p>Total Population: {total_population}</p>

```

```

{/misc}

// Rendered Output
<h1>Information About North America</h1>
<p>Total Population: 565000000</p>

```

9.4.3 迭代

上一个例子创建了一个新的 Dust 片段（和相应上下文）。结果是，新片段的内容可以直接通过被引用对象字面量的属性来访问。以相似方式，Dust 片段还能遍历访问数组。不同于前一个例子中的片段只使用了一次，本例中的片段{#countries} ... {/countries}被使用多次，每当进入其引用的数组中都会被使用一次。

清单 9-23 使用片段遍历数组

```

// Template
{#countries}
{! 通过使用 '$idx' 可以引用迭代循环中的当前位置 !}
{! 可以通过引用 '$len' 获得循环访问对象的大小 !}
<tr>
  <td>{name}</td>
  <td>{population}</td>
  <td>{capital.name}</td>
  <td>{capital.population}</td>
</tr>
{/countries}

// Rendered Output
<tr>
  <td>United States</td>
  <td>319999999</td>
  <td>Washington D.C.</td>
  <td>6600000</td>
</tr>
<tr>
  <td>Mexico</td>
  <td>118000000</td>
  <td>Mexico City</td>
  <td>9000000</td>
</tr>
<tr>
  <td>Canada</td>
  <td>35000000</td>
  <td>Ottawa</td>
  <td>880000</td>
</tr>

```

清单 9-24 展示了模板可以循环包含原始数据类型的数组（不是对象）。每次迭代都可以用{.}语法来引用其自身的值。

清单 9-24 迭代循环包含原始类型数据的数组

```

// Template
<ul>
  {#languages}<li>{.}</li>{/languages}
</ul>

// Rendered Output
<ul>

```



```

<li>English</li>
<li>Spanish</li>
</ul>

```

9.4.4 条件句

Dust 基于简单的真假测试，内置支持了按条件的渲染内容。清单 9-25 中的模板展示了这个概念，通过判断每个国家的 `english` 属性值是不是为真来渲染文字 “Yes” 或 “No”。

清单 9-25 在 Dust 模板内使用条件句

```

// Template
{#countries}
<tr>
  <td>{name}</td>
  <td>{?english}Yes{:else}No{/english}</td>
  {!
    相反的逻辑可以按以下方式使用：
    <td>{^english}No{:else}Yes{/english}</td>
  !}
</tr>
{/countries}

// Rendered Output
<tr>
  <td>United States</td>
  <td>Yes</td>
</tr>
<tr>
  <td>Mexico</td>
  <td>No</td>
</tr>
<tr>
  <td>Canada</td>
  <td>Yes</td>
</tr>

```

■ **注意** 当在模板中应用条件判断时，要理解一个重要的规则，即 Dust 会把“真值”当作一个属性来应用。空字符串、布尔值 `false`、空数组、`null` 和 `undefined` 都会被当作 `false`。数字 `0`、空对象和字符串类型的 “0”、“null”、“undefined” 和 “false” 都会被当作 `true`。

9.4.5 局部模板

Dust 最强大的特性之一是局部模板，让开发者能够在其他模板中引用一个模板。这样一来，复杂的文档可以被分割为更小的组件（如“局部模板”），便于管理和重用。清单 9-26 中的小例子展示了这个过程。

清单 9-26 引用了一个外部模板的 Dust 模板（如“局部模板”）

```

// Main Template
<h1>{report_name}</h1>
<p>Total Population: {misc.total_population}</p>
{>"countries"/}
{!

```

本例中，外部模板 `'countries'` - 由父模板通过名字来引用（使用在模板自身中指定的字符串字面量）。作为替代，通过 Dust 对动态局部模板的支持，外部模板的

`name` 源于继承在模板上下文中特有的值。为达到这一目的，我们将把 `'countries'` 字符串包裹在一对大括号中，就像这样：

```
!}
// "countries" template
{#countries}
<tr>
  <td>{name}</td>
  <td>{population}</td>
  <td>{capital.name}</td>
  <td>{capital.population}</td>
</tr>
{/countries}

// Rendered Output
<h1>Information About North America</h1>
<p>Total Population: 565000000</p>
<tr>
  <td>United States</td>
  <td>Yes</td>
</tr>
<tr>
  <td>Mexico</td>
  <td>No</td>
</tr>
<tr>
  <td>Canada</td>
  <td>Yes</td>
</tr>
```

9.4.6 块

思考一个可能在复杂应用中遇到的普遍场景，即一个应用由多个页面组成。每个页面显示为一个独立的内容集合，它们同时共享相同的元素，如头部和底部。`Dust` 块可以让开发者在单一位置定义这些共享的元素。而后，想要继承它们的模板还有能力在必要的时候重写这些共享元素的内容。

清单 9-27 中的例子帮我们清楚解释了这个概念。示例中展示了定义了整个站点的布局的 `Dust` 模板内容。其中指定了一个默认的页面标题 `{+title}App{/title}`，以占位符的形式放在 `body` 中。

清单 9-27 可在其他模板中继承的 `Dust` 区块

// dust-blocks/views/shared/base.dust

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>{+title}App{/title}</title>
  <link href="/css/style.css" rel="stylesheet">
</head>
<body>
  {+bodyContent/}
</body>
</html>
```

清单 9-28 展示了一个 `Dust` 模板，它继承了清单 9-27 所示的模板。通过引入局部模板的形式嵌入父模板，然后将内容注入定义的 `{+bodyContent/}` 占位符 `<bodyContent>...{/bodyContent}` 中。本例

中，我们的模板没有重写父模板中指定的默认页面标题。

清单 9-28 继承了一个区块的 Dust 模板
// dust-blocks/views/main.dust

```
{>"shared/base"/}  
{<bodyContent>  
  <p>Hello, world!</p>  
{/bodyContent}
```

9.4.7 过滤器

Dust 包含一些内置过滤器，它们能让模板在渲染之前修改其中的值。按照例子中的方式，Dust 会在模板中自动为 HTML 中引用的任意值转码。换句话说，如果上下文中包含一部分如下内容：

```
<script>doBadThings();</script>
```

Dust 会自动把值渲染为

```
&lt;script&gt;doBadThings()&lt;/script&gt;
```

当然，这种特性一般符合预期，一般情况很难遇见不需要这项功能的场景。可以通过使用以下过滤器来禁用它：

```
{content|s}
```

示例中的 |s 过滤器能对当前引用的值禁用自动转码功能。表 9-1 是 Dust 提供的内置过滤器列表。

表 9-1 Dust 提供的内置过滤器

过滤器	描述
S	禁用 HTML 转码
h	强制 HTML 转码
j	强制 JavaScript 转码
u	用 encodeURIComponent()编码
uc	用 encodeURIComponent()编码
js	序列化 JSON 字面量
jp	解析 JSON 字符串

创建自定义过滤器

除了已有的几个主要过滤器，Dust 还提供了自定义功能，让开发者可以很轻易地定制过滤器，如清单 9-29 所示。示例中创建了一个自定义过滤器 formatTS。使用时，此过滤器可以把时间戳转换为可读的格式（如 Jul. 4, 1776）。

清单 9-29 定义自定义 Dust 过滤器
// dust-filters/index.js
var express = require('express');

```

var adaro = require('adaro');
var app = express();
var moment = require('moment');

app.engine('dust', adaro.dust({
  'cache': false,
  'helpers': [
    function(dust) {
      dust.filters.formatTS = function(ts) {
        return moment(ts, 'X').format('MMM. D, YYYY');
      };
    }
  ]
}));

app.set('view engine', 'dust');
app.use('/', express.static('./public'));

app.get('/', function(req, res, next) {
  res.render('main', {
    'events': [
      { 'label': 'Moon Landing', 'ts': -14558400 },
      { 'label': 'Fall of Berlin Wall', 'ts': 626616000 },
      { 'label': 'First Episode of Who\'s the Boss', 'ts': 464529600 }
    ]
  });
});

// dust-filters/views/main.dust (摘录)

<tbody>
  {#events}
  <tr>
    <td>{label}</td>
    <td>{ts|formatTS}</td>
  </tr>
  {/events}
</tbody>

```

9.4.8 上下文辅助器

除了存储数据，Dust 上下文还能存储函数（也被叫作“上下文辅助器”），可以在模板中引用先前传入的函数。按这种方式，Dust 上下文不仅可以被看作原始数据的承载，更可以看作一个视图模型——应用程序的业务逻辑和视图间的中介者，能够以最合适的方式格式化数据信息。

清单 9-30 展示了这个功能，其中应用程序为用户展示服务器列表。每个入口显示一个服务器名以及服务器是否在线的指示器。表头展示系统总体健康状况，由上下文辅助器 `systemStatus` 生成。注意，模板能够像引用其他类型（如对象字面量、数组、数字、字符串）一样引用上下文辅助器。

清单 9-30 Dust 上下文辅助器

```

// dust-context-helpers1/index.js (摘录)

app.all('/', function(req, res, next) {
  res.render('main', {
    'servers': [
      { 'name': 'Web Server', 'online': true },
      { 'name': 'Database Server', 'online': true },
      { 'name': 'Email Server', 'online': false }
    ]
  });
});

```



```

    },
    systemStatus: function(chunk, context, bodies, params) {
      var offlineServers = _.filter(this.servers, { 'online': false });
      return offlineServers.length ? 'Bad' : 'Good';
    }
  });
});

```

// dust-context-helpers1/views/main.dust (摘录)

```

<h1>System Status: {systemStatus}</h1>
<table>
  <thead><tr><th>Server</th><th>Online</th></tr></thead>
  <tbody>
    {#servers}
    <tr>
      <td>{name}</td>
      <td>{?online}Yes{:else}No{/online}</td>
    </tr>
  {/servers}
</tbody>
</table>

```

正如示例所示，每个 Dust 上下文复制器接收四个参数：**chunk**、**context**、**bodies** 和 **params**。让我们来看几个展示其用法的例子。

1. chunk

上下文辅助器的 **chunk** 参数可以用“**chunk**”来获取 Dust 当前模板的一部分。思考清单 9-31 中的示例，其中的上下文辅助器与定义在模板中的默认内容匹配。本例中，**systemStatus()** 上下文辅助器通过调用 **chunk.write()** 方法，覆盖 **chunk** 的默认内容“Unknown”。辅助器可以通过把 **chunk** 作为返回值来指示这个过程。

清单 9-31 Dust 上下文辅助器和默认内容匹配

// dust-context-helpers2/index.js (摘录)

```

app.all('/', function(req, res, next) {
  res.render('main', {
    'servers': [
      { 'name': 'Web Server', 'online': true },
      { 'name': 'Database Server', 'online': true },
      { 'name': 'Email Server', 'online': false }
    ],
    systemStatus: function(chunk, context, bodies, params) {
      if (!this.servers.length) return;
      if (_.filter(this.servers, { 'online': false }).length) {
        return chunk.write('Bad');
      } else {
        return chunk.write('Good');
      }
    }
  });
});

```

// dust-context-helpers2/views/main.dust (摘录)

```

<h1>System Status: {#systemStatus}Unknown{/systemStatus}</h1>

```

2. context

context 参数提供上下文辅助器功能，可以方便地获取定义在模板中的有效上下文片段。清单 9-32

展示了此过程，其为传递的每个 server 数据引用了 `isOnline()` 上下文辅助器。在每一次循环中，`isOnline()` 辅助器都通过 `context.get()` 方法来获取有效片段中的 `online` 属性。

清单 9-32 上下文辅助器通过上下文参数提供可以得到有效片段

```
// dust-context-helpers3/index.js (摘录)

app.all('/', function(req, res, next) {
  res.render('main', {
    'servers': [
      { 'name': 'Web Server', 'online': true },
      { 'name': 'Database Server', 'online': true },
      { 'name': 'Email Server', 'online': false }
    ],
    'systemStatus': function(chunk, context, bodies, params) {
      return _.filter(this.servers, { 'online': false }).length ? 'Bad' : 'Good';
    },
    'isOnline': function(chunk, context, bodies, params) {
      return context.get('online') ? 'Yes' : 'No';
    }
  });
});

// dust-context-helpers3/views/main.dust (摘录)

<h1>System Status: {systemStatus}</h1>
<table>
  <thead><tr><th>Server</th><th>Online</th></tr></thead>
  <tbody>
    {#servers}
    <tr>
      <td>{name}</td>
      <td>{isOnline}</td>
    </tr>
    {/servers}
  </tbody>
</table>
```

3. bodies

想象这样一个场景，大部分的模板内容都是由上下文辅助器决定的。不同于强迫开发者用笨拙的方式拼接字符串，Dust 允许这样的内容保持在模板中它所属的位置，通过上下文辅助器选项决定是否渲染。

清单 9-33 的示例展示了四个不同的 `bodies` 内容被传入 `description()` 上下文辅助器中。通过辅助器的 `bodies` 参数可以引用内容，而后可以通过向 `chunk.render()` 传递合适的值来决定这部分内容是否渲染。

清单 9-33 通过 `bodies` 参数选择是否渲染模板的某个部分

```
// dust-context-helpers4/index.js (摘录)

app.all('/', function(req, res, next) {
  res.render('main', {
    'servers': [
      { 'name': 'Web Server', 'online': true },
      { 'name': 'Database Server', 'online': true },
      { 'name': 'Email Server', 'online': false },
      { 'name': 'IRC Server', 'online': true }
    ]
  });
});
```

```

    },
    'systemStatus': function(chunk, context, bodies, params) {
      return _.filter(this.servers, { 'online': false }).length ? 'Bad' : 'Good';
    },
    'isOnline': function(chunk, context, bodies, params) {
      return context.get('online') ? 'Yes' : 'No';
    },
    'description': function(chunk, context, bodies, params) {
      switch (context.get('name')) {
        case 'Web Server':
          return chunk.render(bodies.web, context);
          break;
        case 'Database Server':
          return chunk.render(bodies.database, context);
          break;
        case 'Email Server':
          return chunk.render(bodies.email, context);
          break;
      }
    }
  });
});
// dust-context-helpers4/index.js (摘录)

<h1>System Status: {systemStatus}</h1>
<table>
  <thead><tr><th>Server</th><th>Online</th><th>Description</th></tr></thead>
  <tbody>
    {#servers}
      <tr>
        <td>{name}</td>
        <td>{isOnline}</td>
        <td>
          {#description}
            {web}
              A web server serves content over HTTP.
            {database}
              A database server fetches remotely stored information.
            {email}
              An email server sends and receives messages.
            {else}
              -
            {/description}
          </td>
        </tr>
      {/servers}
    </tbody>
  </table>

```

4. params

除了从被调用的上下文中引用属性（通过 `context.get()`），上下文辅助器还能访问已经通过模板传入上下文的参数。清单 9-34 的示例向 `formatUptime()` 辅助器传入了每个服务器的 `uptime` 属性。该示例中，在数据填充到模板中之前，辅助器会先把 `params.value` 的值转换为更容易阅读的形式。

清单 9-34 通过 `params` 参数上下文辅助器可接收参数

```

// dust-context-helpers5/index.js (摘录)

app.all('/', function(req, res, next) {

```

```

res.render('main', {
  'servers': [
    { 'name': 'Web Server', 'online': true, 'uptime': 722383 },
    { 'name': 'Database Server', 'online': true, 'uptime': 9571 },
    { 'name': 'Email Server', 'online': false, 'uptime': null }
  ],
  'systemStatus': function(chunk, context, bodies, params) {
    return _.filter(this.servers, { 'online': false }).length ? 'Bad' : 'Good';
  },
  'formatUptime': function(chunk, context, bodies, params) {
    if (!params.value) return chunk.write('-');
    chunk.write(moment.duration(params.value, 'seconds').humanize());
  }
});
// dust-context-helpers5/views/main.dust (摘录)
{#servers}
  <tr>
    <td>{name}</td>
    <td>{?online}Yes{:else}No{/online}</td>
    <td>{#formatUptime value=uptime /}</td>
  </tr>
{/servers}

```

清单 9-35 展示了一个更复杂的例子，解释上下文辅助器参数使用方式。该例中，辅助器 `parseLocation()` 通过接收一个字符串来引用上下文的属性：`value="{name} lives in {location}"`。为了这些引用能够正确显示，参数首先要通过 Dust 的 `helpers.tap()` 方法来计算。

清单 9-35 引用了上下文中属性的参数首先要被求值

// dust-context-helpers6/index.js

```

var express = require('express');
var adaro = require('adaro');
var app = express();
var morgan = require('morgan');
app.use(morgan('combined'));
var engine = adaro.dust();
var dust = engine.dust();

app.engine('dust', engine);

app.set('view engine', 'dust');
app.use('/', express.static('./public'));

app.all('/', function(req, res, next) {
  res.render('main', {
    'people': [
      { 'name': 'Joe', 'location': 'Chicago' },
      { 'name': 'Mary', 'location': 'Denver' },
      { 'name': 'Steve', 'location': 'Oahu' },
      { 'name': 'Laura', 'location': 'Nashville' }
    ],
    'parseLocation': function(chunk, context, bodies, params) {
      var content = dust.helpers.tap(params.value, chunk, context);
      return chunk.write(content.toUpperCase());
    }
  });
});

```



```
app.listen(8000);

// dust-context-helpers6/views/main.dust

{#people}
  <li>{#parseLocation value="{name} lives in {location}" /}</li>
{/people}
```

5. 异步上下文辅助器

辅助器函数让 Dust 更为强大和灵活。它们使上下文对象作为视图模型——一个在应用的业务逻辑和用户界面之间的智能桥梁，在将其传递到一个或多个视图前，承载了获取数据以及按特定用例格式化功能。尽管你看到它是如此有用，但对于这些辅助函数的强大作用而言，我们仅仅知晓了其皮毛。

除了直接返回数据，Dust 辅助函数也可以返回异步数据。清单 9-36 显示了这样一个过程。示例中重建了两个上下文辅助器，分别为 `cars()` 和 `trucks()`。前者返回一个数组，后者返回一个 `promise` 对象，其 `resolve` 了一个数组。从模板的角度来看，这两个函数的需求相同。

清单 9-36 能返回 promise 的辅助函数

// dust-promise1/index.js (摘录)

```
app.get('/', function(req, res, next) {
  res.render('main', {
    'cars': function(chunk, context, bodies, params) {
      return ['Nissan Maxima', 'Toyota Corolla', 'Volkswagen Jetta'];
    },
    'trucks': function(chunk, context, bodies, params) {
      return new Promise(function(resolve, reject) {
        resolve(['Chevrolet Colorado', 'GMC Canyon', 'Toyota Tacoma']);
      });
    }
  });
});
```

// dust-promise1/views/main.dust (摘录)

```
<h1>Cars</h1>
<ul>{#cars}<li>{.}</li>{/cars}</ul>
<h2>Trucks</h2>
<ul>{#trucks}<li>{.}</li>{/trucks}</ul>
```

对于按条件显示内容，Dust 还提供了便捷的方法，如清单 9-37 所示，示例中的 `promise` 被拒绝。

清单 9-37 处理被拒绝的 promise

// dust-promise2/index.js (摘录)

```
app.get('/', function(req, res, next) {
  res.render('main', {
    'cars': function(chunk, context, bodies, params) {
      return ['Nissan Maxima', 'Toyota Corolla', 'Volkswagen Jetta'];
    },
    'trucks': function(chunk, context, bodies, params) {
      return new Promise(function(resolve, reject) {
        reject('Unable to fetch trucks.');
```

```

    });
});

// dust-promise2/views/main.dust (摘录)

<h1>Cars</h1>
<ul>{#cars}<li>{.}</li>{/cars}</ul>
<h2>Trucks</h2>
<ul>{#trucks}
  <li>{.}</li>
  {error}
</ul>{/trucks}</ul>

```

错误。无法获取卡车列表。

有很多理由可认为以 `promise` 的形式向模板返回信息的能力很有用，但当这项功能和 `Dust` 的流式传输接口结合使用时，情况就会更为有趣。思考清单 9-38 中的内容，它和前一个示例很像。然而这里利用了 `Dust` 的流式接口，它不是等待整个模板渲染过程完成，而是在模板正在渲染时把一部分模板下推到客户端。

清单 9-38 当数据可用时向客户端流式传输模板

```

// dust-promise2/index.js

var Promise = require('bluebird');
var express = require('express');
var adaro = require('adaro');, ,
var engine = adaro.dust();
var dust = engine.dust;
app.engine('dust', engine);
app.set('view engine', 'dust');
app.use('/', express.static('./public'));

app.get('/', function(req, res, next) {
  dust.stream('views/main', {
    'cars': ['Nissan Maxima', 'Toyota Corolla', 'Volkswagen Jetta'],
    'trucks': function(chunk, context, bodies, params) {
      return new Promise(function(resolve, reject) {
        setTimeout(function() {
          resolve(['Chevrolet Colorado', 'GMC Canyon', 'Toyota Tacoma']);
        }, 4000);
      });
    }
  }).pipe(res);
});

app.listen(8000);

```

取决于示例中模板的复杂性，在用户体验上，这种方法的效果往往引人注目。不同于强迫用户在他们能继续使用前等待整个页面加载完成，这种方法让模板只要可用就下推到客户端。这样做的结果是，用户访问应用时感知到的延迟明显减少。

9.4.9 Dust 辅助器

在前面的小节中，我们探索了上下文对象如何通过使用上下文辅助器来引入与特定视图层相关的逻辑。以相似的方式，`Dust` 可以让辅助器在全局层面定义，使其在包含其上下文中显示定义之前就可以使用。利用这些功能，在面对更严格的、无逻辑的模板时，开发者能够更容易地解决遇到的挑战。

清单 9-39 显示了一段 JSON 数据的摘录，它会在本节的余下示例中引用。

清单 9-39 传递给 Dust 模板的 JSON 数据片段

// dust-logic1/people.json (摘录)

```
[{
  "name": "Joe", "location": "Chicago", "age": 27,
  "education": "high_school", "employed": false, "job_title": null
}, {
  "name": "Mary", "location": "Denver", "age": 35,
  "education": "college", "employed": true, "job_title": "Chef"
}]
```

1. 逻辑辅助器

清单 9-40 显示了逻辑辅助器@eq 的用法，这个功能使我们能够在两个指定的键和值之间执行严格比较。本例中，前一个值 job_title 在当前上下文中引用了一个属性。第二个值“Chef”在模板中定义为一个字面量值。

清单 9-40 使用 Logic 辅助器按条件显示内容

// dust-logic1/views/main.dust (摘录)

```
{#people}
  {@eq key=job_title value="Chef"}
    <p>{name} is a chef. This person definitely knows how to cook.</p>
  {:else}
    <p>{name} is not a chef. This person may or may not know how to cook.</p>
  {/eq}
{/people}
```

学习这些内容后，想象这样一个场景：我们想要在两个数字间执行严格相等比较，其中一个数字被作为上下文中的一个属性引用，另一个数字在模板中被指定为一个字面量。为了做比较，我们必须把字面量值转换为合适的类型，如清单 9-41 所示。

清单 9-41 把字面量值转换为需要的类型

```
{#people}
  {@eq key=age value="27" type="number"}
    <p>{name} is 27 years old.</p>
  {/eq}
{/people}
```

Dust 提供了大量的逻辑辅助器用于做简单的比较。表 9-2 展示了它们的名字和描述。

表 9-2 Dust 提供的逻辑辅助器

逻辑辅助器	描述
@eq	严格相等
@ne	非严格相等
@gt	大于
@lt	小于
@gte	大于或等于
@lte	小于或等于

2. Switch 表达式

我们可以通过常用的辅助器@select 来模拟 switch (...)语句，使我们能够基于一个特定值来指定多种内容（见清单 9-42）。

清单 9-42 使用@select 辅助器模拟 switch 语句

```
{@gte key=age value=retirement_age}
  <p>{name} has reached retirement age.</p>
  {else}
    <p>
      {@select key=job_title}
        {@eq value="Chef"}Probably went to culinary school, too.{/eq}
        {@eq value="Professor"}Smarty pants.{/eq}
        {@eq value="Accountant"}Good with numbers.{/eq}
        {@eq value="Astronaut"}Not afraid of heights.{/eq}
        {@eq value="Pilot"}Travels frequently.{/eq}
        {@eq value="Stunt Double"}Fearless.{/eq}
        {! @none serves as a `default` case !}
        {@none}Not sure what I think.{/none}
      {/select}
    </p>
  {/gte}
```

3. 迭代辅助器

为解决在迭代中经常遇到的问题，Dust 提供了三个有用的辅助器。清单 9-43 展示了辅助器@sep 的使用。通过这个辅助器，我们能够定义除了最后一次的每次渲染内容。

清单 9-43 使用@sep 辅助器在迭代中忽略最后一次内容

```
// dust-logic1/views/main.dust (摘录)
{#people}{name}{@sep}, {/sep}{/people}

// output
Joe, Mary, Wilson, Steve, Laura, Tim, Katie, Craig, Ryan
```

Dust 共提供了三个辅助器来解决这类迭代问题，如表 9-3 所示。

表 9-3

迭代辅助器

迭代辅助器	描述
@sep	除最后一次外，为每次迭代渲染内容
@first	仅为第一次迭代渲染内容
@last	仅为最后一次迭代渲染内容

4. 数学表达式

使用 Dust 的@math 辅助器，模板可以通过数学表达式的结果来调整其内容。这种调整可以发生在两种方式之一。清单 9-44 展示了第一种方法，数学表达式的结果直接在模板中引用。清单 9-45 展示了第二种方法，基于调用@math 辅助器所得到的返回结果条件渲染模板内容。

清单 9-44 直接引用数学表达式的结果

```
// dust-logic1/views/main.dust (摘录)
{#people}
```



```

    {@lt key=age value=retirement_age}
      <p>{name} will have reached retirement age in
      {@math key=retirement_age method="subtract" operand=age /} year(s).</p>
    {/lt}
  {/people}

```

清单 9-45 基于@math 辅助器调用结果条件渲染内容

```
// dust-logic1/views/main.dust (摘录)
```

```

{#people}
  {@lt key=age value=retirement_age}
    {@math key=retirement_age method="subtract" operand=age}
      {@lte value=10}{name} will reach retirement age fairly soon.{/lte}
      {@lte value=20}{name} has quite a ways to go before they can retire.{/lte}
      {@default}{name} shouldn't even think about retiring.{/default}
    {/math}
  {/lt}
{/people}

```

Dust 的@math 辅助器支持的多种方法包括 add、subtract、multiply、divide、mod、abs、floor 和 ceil。

5. 全部打印上下文内容

Dust 的@contextDump 辅助器可以快速渲染当前上下文对象（以 JSON 格式），这在开发过程中很有用。这里展示了其用法示例：

```
{#people}<pre>{@contextDump /}</pre>{/people}
```

6. 自定义辅助器

本章的前文中，你了解了如何创建上下文辅助器，且通过该辅助器能够为上下文对象扩展自定义功能。以相同的方式，还能在全局层面创建自定义 Dust 辅助器。清单 9-46 展示了其如何应用。

清单 9-46 创建和使用自定义 Dust 辅助器

```
// dust-logic1/index.js (摘录)
```

```

dust.helpers.inRange = function(chunk, context, bodies, params) {
  if (params.key >= params.lower && params.key <= params.upper) {
    return chunk.render(bodies.block, context);
  } else {
    return chunk;
  }
}

```

```
// dust-logic1/views/main.dust (摘录)
```

```

{#people}
  {@gte key=age value=20}
    {@lte key=age value=29}<p>This person is in their 20's.</p>{/lte}
  {/gte}
  {@inRange key=age lower=20 upper=29}<p>This person is in their 20's. </p>{/inRange}
{/people}

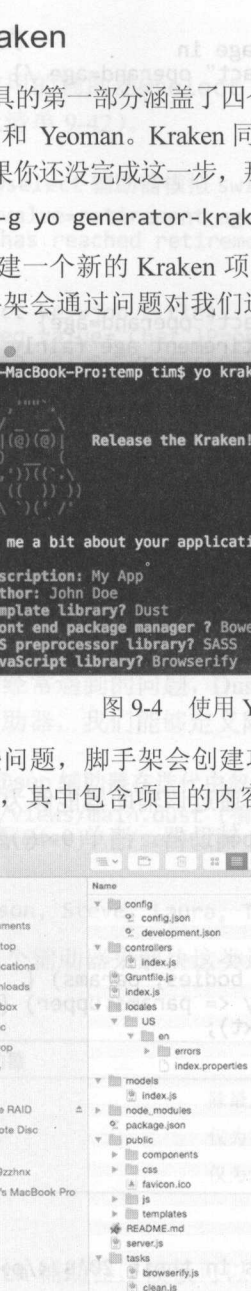
```

本例中的模板创建了一个循环，在其上下文中迭代了每个人。对于某个人恰好落在二十几岁的年龄段，就会显示出一条信息。首先，在逻辑辅助器@gte 和@lt 的结合下显示这条信息。随后，这条信息再次显示，在全局层面使用自定义@inRange 辅助器。现在你已经对 Kraken 依赖的这些基础组件很熟悉了，让我们继续提步向前，创建我们第一个真正的 Kraken 应用。

[illegible]

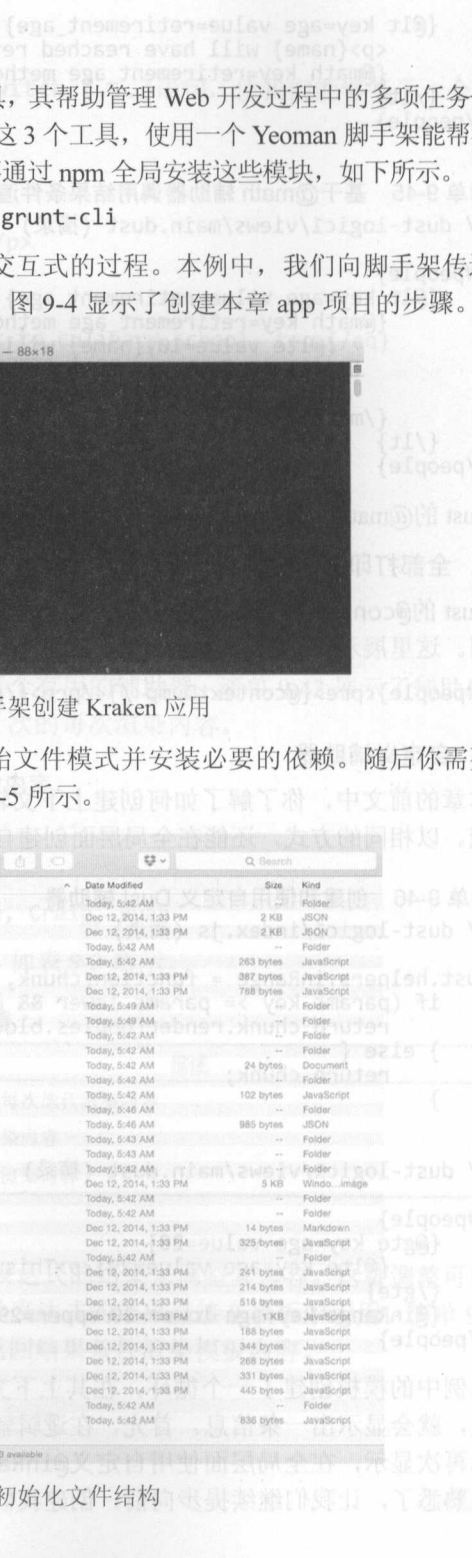
一旦你答
新的 app

一旦你答完这些问题，脚手架会创建项目的初始文件模式并安装必要的依赖。随后你需要找到一个新的 **app** 文件夹，其中包含项目的内容，如图 9-5 所示。



项目

用工具
依赖于
你需要
power
一个
提示、
mp - bas
an 脚
的初
如图！
ap
目 app



要按

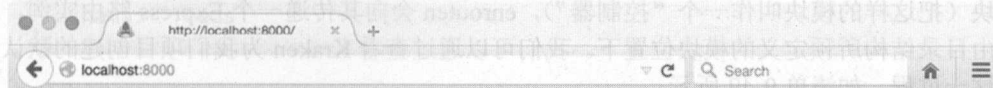
Kraken 的 Yeoman 脚手架自动执行创建一个新 Express 应用的过程, 该应用使用本章前面提到的模块化方式组织。我们可以用清单 9-47 所示的当前状态立刻启动项目。随后, 项目可以通过一个本地地址访问, 如图 9-6 所示。

清单 9-47 第一次启动项目

```
$ npm start

> app@0.1.0 start /Users/tim/temp/app
> node server.js
```

```
Server listening on http://localhost:8000
Application ready to serve requests.
Environment: development
```



Hello, index!

图 9-6 第一次在浏览器中查看项目

正如你看到的, 为了使用一些有益的中间件模块 (如 `cookieParser`、`session` 等), 我们的项目已经预先配置好了。为深入了解这一切是如何结合在一起的, 清单 9-48 显示了项目的 `index.js` 脚本内容。

清单 9-48 新项目的 `index.js` 脚本内容

```
// app/index.js

var express = require('express');
var kraken = require('kraken-js');

var options, app;
/*
 * 创建和配置应用。同样通过测试导出应用程序示例。
 * 查看 https://github.com/krakenjs/kraken-js#options 获得更多配置选项。
 */
options = {
  onconfig: function (config, next) {
    /*
     * 这里可增加任意 config 的设定或重写。'config' 是 'config' 配置对象的预置。
     * 参见 (https://github.com/krakenjs/config/).
     */
    next(null, config);
  }
};

app = module.exports = express();

app.use(kraken(options));
app.on('start', function () {
  console.log('Application ready to serve requests.');
```

在这里看到的 `kraken-js` 模块, 不过是一个标准的 Express 中间件库。然而, 不同于仅仅使

用几个额外的功能增强 Express, Kraken 负责配置一个完整的 Express 应用。它在很多其他模块的辅助下完成这项职责,其中包括本章已经提到的那些模块: `confit`、`meddleware`、`enrouten` 和 `adaro`。

如清单 9-48 所示, Kraken 传递了一个配置对象,该对象包含一个 `onconfig()` 回调函数,其会在 Kraken 已经处理 `confit` 初始化工作后调用。我们可能不想直接在项目的 JSON 配置文件中定义,在这里我们可以提供任意时刻对配置的重写。本例中并未做这样的重写。

1. 控制器、模型和测试

在本章中的“模式化路由注册”一节中,我们探索了 `enrouten` 是如何为经常按杂乱无章组织的 Express 路由整顿秩序的。默认情况下,一个新的 Kraken 项目使用 `enrouten` 的目录配置选项启动,通过递归扫描指定文件夹的内容,搜寻导出函数是接受单一参数的模块(如 `router`)。对于每个其找到的模块(把这样的模块叫作一个“控制器”),`enrouten` 会向其传递一个 Express 路由实例,其已经挂载到由目录结构所预定义的模块位置下。我们可以通过查看 Kraken 为我们项目创建的默认控制器来学习这一过程,如清单 9-49 所示。

清单 9-49 项目的默认控制器

```
// app/controllers/index.js
```

```
var IndexModel = require('../models/index');
```

```
module.exports = function (router) {
```

```
    var model = new IndexModel();
```

```
    /**
```

```
     * 当我们通过 http://localhost:8000 访问 app 时的默认路由
```

```
    */
```

```
    router.get('/', function (req, res) {
        res.render('index', model);
    });
```

```
};
```

除了为项目创建默认控制器, Kraken 还会处理对相应模型 `IndexModel` 的创建,如清单 9-49 所示。我们会对 Kraken 与模型的关系做简要讨论,但首先详细解释一下创建一个我们自己的控制器的方式。

第 3 章中讨论了 Yeoman, 其脚手架为开发者提供了子命令功能,可以很好地对项目创建初始化做扩展。Kraken 的 Yeoman 脚手架利用了这一点,并提供控制器子命令,从而使我们快速创建控制器。按照实例中的方式,我们创建一个负责管理 RSS 反馈集合的控制器。

```
$ yo kraken:controller feeds
```

在你使用脚手架的控制器子命令指定路径 `feeds` 后,脚手架自动创建以下五个新文件。

- `controllers/feeds.js`: 控制器
- `models/feeds.js`: 模型
- `test/feeds.js`: 测试套件
- `public/templates/feeds.dust`: Dust 模板

- `locales/US/en/feeds.properties`: 国际化设置

目前来说, 我们首先集中于前三个文件, 先从模型开始。在下一节中, 我们会看看附带的 Dust 模板和国际化设置文件。

(1) 模型

清单 9-50 显示了项目更新后的反馈模型初始状态。如果你期待看到某些复杂精巧的代码, 那么你可能会失望。正如你看到的, 这个文件仅提供一个基本的桩, 用于替换为我们自己的持久层。

清单 9-50 反馈模型的初始内容

```
// models/feeds.js
```

```
module.exports = function FeedsModel() {
  return {
    name: 'feeds'
  };
};
```

不同于很多其他“全栈”框架试图为开发者提供处理所有能想到问题的工具(包括数据持久化), Kraken 采取了一种不尝试重复发明轮子的极简方法。这种方法承认开发者会使用很多支持良好的库来管理数据持久化。本书中也涵盖了两个: Knex/Bookshelf 和 Mongoose。

按照示例中的方式, 让我们更新此模块以便于它导出一个 Bookshelf 模型。该模型能够从 SQLite 数据库中的反馈数据表中存取信息。清单 9-51 展示了反馈模型更新后的内容。

清单 9-51 使用 Knex/Bookshelf 更新反馈模型

```
// models/feeds.js
```

```
var bookshelf = require('../lib/bookshelf');
var Promise = require('bluebird');
var feedRead = require('feed-read');
```

```
var Feed = bookshelf.Model.extend({
  'tableName': 'feeds',
  'getArticles': function() {
    var self = this;
    return Promise.fromNode(function(callback) {
      feedRead(self.get('url'), callback);
    });
  }
});
```

```
module.exports = Feed;
```

■ 注意 清单 9-51 的示例中更新后的模型假设你已经对 Knex 和 Bookshelf 库有所熟悉, 其中几个步骤是必须对其配置, 否则你可能需要阅读第 12 章。不管怎样, 在这里, 本章的 app 项目提供了完整的功能代码展示。

(2) 控制器

清单 9-52 显示了项目新反馈控制器的初始内容。它和原始控制器共同组成了我们的项目。这个控制器中引用了相关模型, 如上个例子所示, 该模型 Kraken 已经很方便地为我们创建好了。

清单 9-52 反馈控制器的初始内容

```
// controllers/feeds.js

var FeedsModel = require('../models/feeds');

/**
 * @url http://localhost:8000/feeds
 */
module.exports = function (router) {
  var model = new FeedsModel();
  router.get('/', function (req, res) {
  });
};
```

在默认状态下，反馈控制器所完成的工作很少。让我们来更新这个控制器，为其引入一些额外路由，由此允许客户端和我们应用中的反馈模型交互。更新后的反馈控制器如清单 9-53 所示。

清单 9-53 更新后的反馈控制器

```
var Feed = require('../models/feeds');

module.exports = function (router) {

  router.param('feed_id', function (req, res, next, id) {
    Feed.where({
      'id': id
    }).fetch({});
    'require': true
  }).then(function (feed) {
    req.feed = feed;
    next();
  }).catch(next);
});

/**
 * @url http://localhost:8000/feeds
 */
router.route('/')
  .get(function (req, res, next) {
    return Feed.where({})
      .fetchAll()
      .then(function (feeds) {
        if (req.accepts('html')) {
          return res.render('feeds', {
            'feeds': feeds.toJSON()
          });
        } else if (req.accepts('json')) {
          return res.send(feeds);
        } else {
          throw new Error('Unknown `Accept` value: ' + req.headers.accept);
        }
      })
      .catch(next);
  });

/**
 * @url http://localhost:8000/feeds/:feed_id
 */
router.route('/:feed_id')
  .get(function (req, res, next) {
    res.send(req.feed);
  });
};
```

```

/**
 * @url http://localhost:8000/feeds/:feed_id/articles
 */
router.route('/:feed_id/articles')
  .get(function(req, res, next) {
    req.feed.getArticles()
      .then(function(articles) {
        res.send(articles);
      })
      .catch(next);
  });
};

```

恰当地做了这些更新后，客户端目前拥有这样的能力：

- 列表提要
- 对指定反馈提取信息
- 从指定反馈提取文章

下一节中，我们会看一看 Kraken 为这部分所创建的测试套件。这个测试套件可以帮我们验证定义的路由是否按预期工作。

(3) 测试套件

清单 9-54 展示了 Kraken 为控制器创建的测试套件初始内容。这里定义了一段测试，其依赖于 SuperAgent 的扩展 SuperTest，作为发送 HTTP 请求的简单库。

清单 9-54 反馈控制器的测试套件

```

// test/feeds.js

var kraken = require('kraken-js');
var express = require('express');
var request = require('supertest');

describe('/feeds', function() {
  var app, mock;

  beforeEach(function(done) {
    app = express();
    app.on('start', done);
    app.use(kraken({
      'basedir': process.cwd()
    }));
    mock = app.listen(1337);
  });

  afterEach(function(done) {
    mock.close(done);
  });

  it('should say "hello"', function(done) {
    request(mock)
      .get('/feeds')
      .expect(200)
      .expect('Content-Type', /html/)
      .expect(/"name": "index"/)
      .end(function(err, res) {
        done(err);
      });
  });
});

```

```
});
});
});
```

本例中向应用的/**feeds** 端点发送了 GET 请求，并做了如下断言。

- 服务器应该返回 HTTP 状态码 200。
- 服务器应该返回一个包含字符串格式 html 的 **Content-Type** 头。
- 相应体需要包含字符串 **"name": "index"**。

对于我们新控制器的更新代码，这几个断言则不再适用。让我们用几个相应的测试做替代。清单 9-55 展示了测试套件的更新后内容。

清单 9-55 反馈控制器测试套件的更新内容

```
// test/feeds/index.js
```

```
var assert = require('assert');
var kraken = require('kraken-js');
var express = require('express');
var request = require('supertest');
```

```
describe('/feeds', function() {
```

```
  var app, mock;
```

```
  beforeEach(function(done) {
    app = express();
    app.on('start', done);
    app.use(kraken({ 'basedir': process.cwd() }));
    mock = app.listen(1337);
  });
```

```
  afterEach(function(done) {
    mock.close(done);
  });
```

```
  it('should return a collection of feeds', function(done) {
    request(mock)
      .get('/feeds')
      .expect('Content-Type', /json/)
      .expect(200)
      .end(function(err, res) {
        if (err) return done(err);
        assert(res.body instanceof Array, 'Expected an array');
        done();
      });
  });
```

```
  it('should return a single feed', function(done) {
    request(mock)
      .get('/feeds/1')
      .expect('Content-Type', /json/)
      .expect(200)
      .end(function(err, res) {
        if (err) return done(err);
        assert.equal(typeof res.body.id, 'number',
          'Expected a numeric `id` property');
        done();
      });
  });
```

```
  it('should return articles for a specific feed', function(done) {
```



```

request(mock)
  .get('/feeds/1/articles')
  .expect('Content-Type', /json/)
  .expect(200)
  .end(function(err, res) {
    if (err) return done(err);
    assert(res.body instanceof Array, 'Expected an array');
    done();
  });
});

```

```
});
```

更新后的测试套件现在包含三个测试，用于验证新控制器的每个路由功能。考虑第一段测试脚本，例如，它会向应用的/feeds 端点发送 GET 请求并做如下断言。

- 服务器应该返回 HTTP 状态码 200。
- 服务器应该返回一个包含 json 字符串的 Content-Type 头。
- 服务器应该返回一到多个数组形式的结果。

注意 回顾应用的 Feed 模型是通过使用 Knex 和 Bookshelf 库来创建的。引用在项目中的数据源自于一个 Knex 的“种子”文件(seeds/developments/00-feeds.js)。我们可以用它来使用样例数据计算得到我们的数据库。在任何时候，项目的 SQLite 数据库都能通过在命令行运行 \$ grunt reset-db 被重置为初始状态。如果你对这个概念还不熟悉，请阅读第 12 章。

图 9-7 显示了项目 Grunt 的测试任务调用时，打印到控制台的输出。

```

Tims-MacBook-Pro:app tim$ grunt test
Running "jshint:files" (jshint) task
>> 6 files lint free.

Running "mochacli:src" (mochacli) task

/feeds
127.0.0.1 - - [11/Jun/2015:13:21:28 +0000] "GET /feeds HTTP/1.1" 200 172 "-" "-"
  ✓ should return a collection of feeds (86ms)
127.0.0.1 - - [11/Jun/2015:13:21:28 +0000] "GET /feeds/1 HTTP/1.1" 200 73 "-" "-"
  ✓ should return a single feed
127.0.0.1 - - [11/Jun/2015:13:21:29 +0000] "GET /feeds/1/articles HTTP/1.1" 200 15946 "-" "-"
  ✓ should return articles for a specific feed (244ms)

/
127.0.0.1 - - [11/Jun/2015:13:21:29 +0000] "GET / HTTP/1.1" 200 244 "-" "-"
  ✓ should say "hello"

4 passing (1s)

Done, without errors.

```

图 9-7 运行测试套件

2. 国际化和本地化

对希望在多种不同市场上广泛使用的应用来说，创建有能力应对多语言多区域的应用是非常重要的。Kraken 为其提供了内置的支持。本节中，我们看一看完成国际化和本地化的两个步骤，以及它们怎样应用在服务端模板渲染的 Kraken 应用上下文中。

国际化（经常简称为 i18n）是指开发应用时要支持多种区域和方言的行为。在实践中，通过避免直接使用在应用的模板中的特定于本地的词汇、句子以及标识（如货币标识）来实现。取而代之，

先使用占位符，直到模板请求时，基于用户请求中的定位或者设置来计算得到相应的文字。思考清单 9-56 中的示例，其中展示了 Dust 模板用于渲染本章 app 项目的主页。

清单 9-56 负责渲染 app 项目首页的 Dust 模板

```
// app/public/templates/index.dust

{>"layouts/master" />}

{<body>

  <div class="panel panel-default">
    <div class="panel-heading">
      <h3 class="panel-title">{@pre type="content" key="greeting" /></h3>
    </div>
    <div class="panel-body">
      <form method="post" action="/sessions">
        <div class="form-group">
          <label>{@pre type="content" key="email_address" /></label>
          <input type="email" name="email" class="form-control">
        </div>
        <div class="form-group">
          <label>{@pre type="content" key="password" /></label>
          <input type="password" name="password" class="form-control">
        </div>
        <button type="submit" class="btn btn-primary">
          {@pre type="content" key="submit" />
        </button>
      </form>
    </div>
  </div>
</div>

{</body>}
```

这里使用的基本语法，你应该很熟悉，它基于本章前面提到的关于 Dust 的教程。正如你看到的，不同于直接嵌入内容，这个模板依赖于 Kraken 所提供的特定 Dust 辅助器@pre。使用这个辅助器，我们能够引用单独存储的内容。相应内容文件如清单 9-57 所示。

清单 9-57 Dust 模板的相应内容文件

```
// app/locales/US/en/index.properties
# Comments are supported
greeting=Welcome to Feed Reader
submit=Submit
email_address=Email Address
password=Password

// app/locales/ES/es/index.properties
greeting=Bienvenida al Feed Reader
submit=Presentar
email_address=Correo Electrónico
password=Contraseña
```

注意 留意示例中模板的路径 public/templates/index.dust，以及相应内容属性文件的路径 locales/US/en/index.properties 和 locales/ES/es/index.properties。对于内容属性文件，Kraken 配置了一对 Dust 模板。例如，在一对一的基础上，基于它们的路径和文件来匹配它们。

在创建项目过程中，相比国际化（i18n），有能力支持本地化内容的注入是首要关心的。本地化（l10n）通过特定于本地以及方言的内容文件来完成。清单 9-58 中的控制器显示了 Kraken 如何帮助开发者开发满足用户特殊需求的产品。

清单 9-58 指定地点版本的主页服务

```
// app/controllers/index.js
```

```
module.exports = function(router) {

  /**
   * 当通过 http://localhost:8000 访问 app 时，负责处理请求的默认路由
   */
  router.get('/', function(req, res) {
    res.locals.context = {
      'locality': { 'language': 'es', 'country': 'ES' }}
    res.render('index');
  });
};
```

本示例为清单 9-49 中控制器的更新版本，其负责渲染应用的主页。这里指定了国家和语言，通过把它们赋值给到达的 Express 相应对象中的 `locals.context` 属性，用于定位内容文件。如果没指定这样的值，Kraken 的默认行为是使用英文。渲染的英文和西班牙版本的模板分别如图 9-8 和图 9-9 所示。

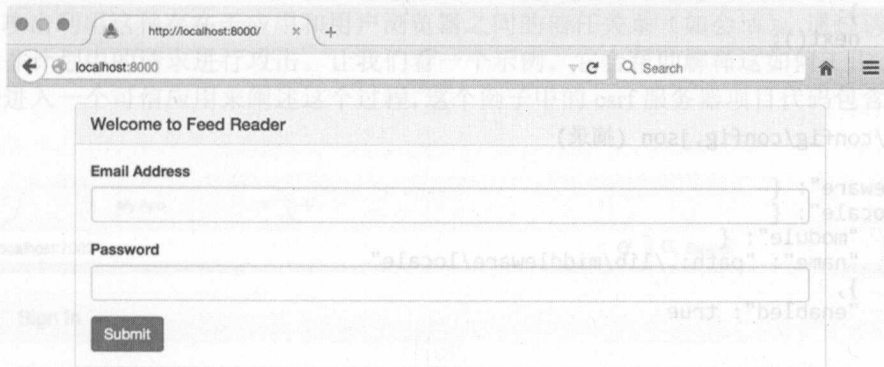


图 9-8 英语版本的应用主页

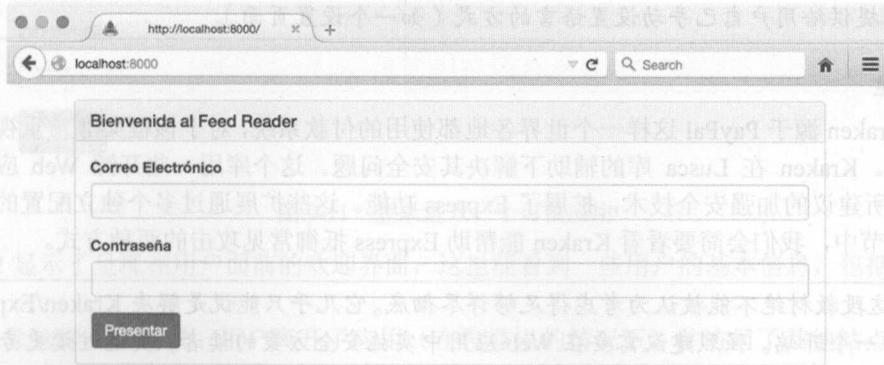


图 9-9 检测到位置后的西班牙版本应用主页

检测地区

清单 9-58 中的示例演示了指定区域的设置可以手动赋值给进入的请求过程，然而没有展示出的内容是用户期望的本地化设置能自动被检测到。

清单 9-59 展示了基于 HTTP 请求头 `accept-language` 来决定本地化的简单方法。这个示例中，我们从路由中移除了检测用户位置的处理逻辑，并将其移动到一个更合适的地方——一个中间件函数在每次请求到达时被调用。

清单 9-59 基于 HTTP 请求头的 `accept-language` 值来检测地区

```
// app/lib/middleware/locale.js

var acceptLanguage = require('accept-language');

/**
 * Express middleware function that automatically determines locality based on the value
 * of the `accept-language` header.
 */
module.exports = function() {

  return function(req, res, next) {
    var locale = acceptLanguage.parse(req.headers['accept-language']);
    res.locals.context = {
      'locality': { 'language': locale[0].language, 'country': locale[0].region }
    };
    next();
  };
};

// app/config/config.json (摘录)

"middleware": {
  "locale": {
    "module": {
      "name": "path:../lib/middleware/locale"
    },
    "enabled": true
  }
}
```

注意 尽管 `accept-language` 请求头很有用，但它并不能总是正确地反映用户期望的地区。始终要保证提供给用户自己手动设置语言的方式（如一个设置页面）。

3. 安全

由于 Kraken 源于 PayPal 这样一个世界各地都使用的付款系统，对于该框架非常重视安全来说应该不足为奇。Kraken 在 Lusca 库的辅助下解决其安全问题。这个库用一些开源 Web 应用安全项目（OWASP）所建议的加强安全技术，扩展了 Express 功能。这些扩展通过多个独立配置的中间件模块来配置。本节中，我们会简要看看 Kraken 能帮助 Express 抵御常见攻击的两种方式。

注意 这段教材绝不能被认为考虑得足够详尽彻底。它几乎只能说是解决 Kraken/Express 应用中安全问题的一个开端。强烈建议需要在 Web 应用中实施安全方案的读者，要通过读更专业的、完全致力于该主题下的书来进一步钻研。

(1) 防御跨站请求伪造攻击

为理解跨站请求伪造 (CSRF) 攻击的基本前提, 理解大部分 Web 应用为用户授权的机制很重要: 基于 cookie 的校验。这个过程在图 9-10 中阐明了。

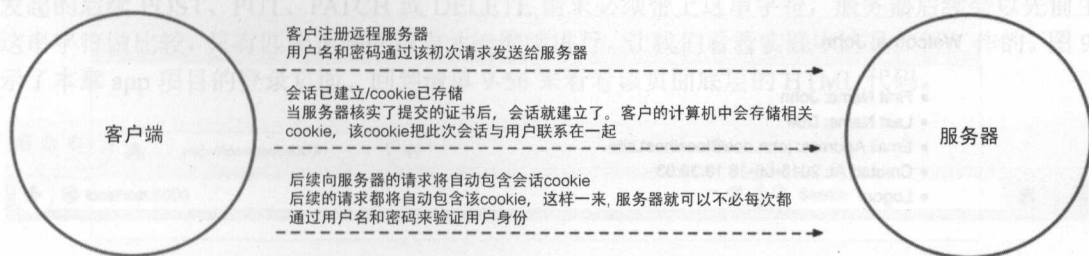


图 9-10 基于 cookie 的授权过程

在典型的场景中, 用户需要向 Web 应用提交个人的信用信息, 用于后续和文件中保存的信息做比较。假设信用认证无效, 服务器实际上会启动一个新的会话, 用于记录用户成功登录的尝试性连接。属于这个会话的唯一的标识符随后会以 cookie 的形式被传输给用户, 自动存储在用户的浏览器中。浏览器发出的后续请求会自动附加存储在 cookie 中的信息, 允许应用寻找匹配到的会话记录。这样一来, 应用就有能力在不需要用户每次重新提交用户名密码来验证身份信息。

CSRF 攻击利用这种存在于应用和用户浏览器之间的信任关系 (如会话), 通过诱骗用户向应用提交一个非计划中的请求进行攻击。让我们看一个示例, 它会帮助解释这如何工作。图 9-11 通过用户登录进入一个可信应用来阐述这个过程, 这个例子中的 csrf 服务器项目代码包含在本章的源代码中。

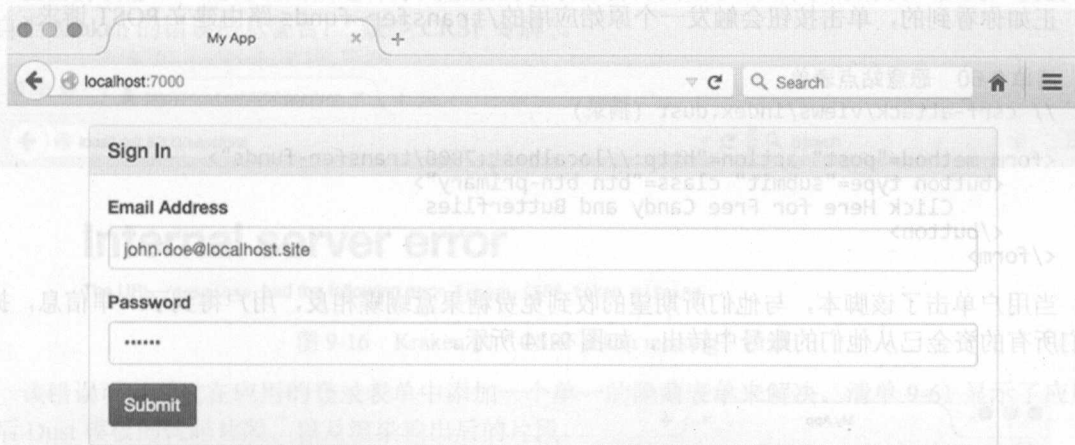


图 9-11 登录进入一个可信应用

图 9-12 显示了呈现在用户面前的欢迎界面, 这里能看到一些用户的基本信息, 包括他的名字以及账号。

此时想象这样一个场景: 用户离开了应用 (在没登出的情况下) 并访问了其他站点, 在用户不知道的情况下试图恶意攻击 (见图 9-13)。这个恶意站点的复制在本章的 csrf-attack 项目中。在该示例中, 恶意站点以免费糖果及蝴蝶来引诱用户单击按钮。

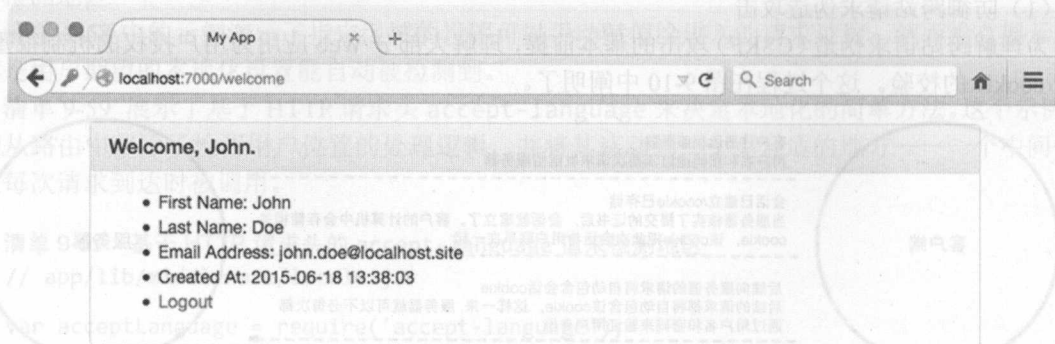


图 9-12 成功尝试登录



图 9-13 试图说服用户单击该按钮的恶意 Web 站点

清单 9-60 展示了该恶意站单的一段 HTML 代码，这段代码用来说明当用户单击按钮后会发什么。正如你看到的，单击按钮会触发一个原始应用的 `/transfer-funds` 路由建立 POST 请求。

清单 9-60 恶意站点表单

// csrf-attack/views/index.dust (摘录)

```
<form method="post" action="http://localhost:7000/transfer-funds">
  <button type="submit" class="btn btn-primary">
    Click Here for Free Candy and Butterflies
  </button>
</form>
```

当用户单击了该脚本，与他们所期望的收到免费糖果盒蝴蝶相反，用户得到了一串信息，提示他们所有的资金已从他们的账号中转出，如图 9-14 所示。

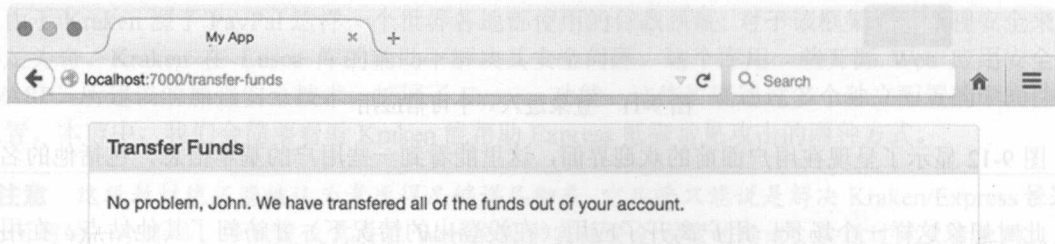


图 9-14 成功的 CSRF 攻击

几个不同的步骤可以用来防御这种性质的攻击。Kraken 防御这个问题的方式是将其引用为“同步令牌模式”。按照这个方法，为每个到达的请求都会生成一个随机字符串，客户端后续能够从模板上下文或响应头的一部分中获取该令牌。重要的一点是，这串字符串不能被存到 cookie 中。由客户端发起的后续 POST、PUT、PATCH 或 DELETE 请求必须带上这串字符，服务器后续会以先前生成的这串字符做比较，只有匹配正确的请求才能继续进行。让我们看看实践中这是如何工作的。图 9-15 显示了本章 app 项目的登录页面。回到清单 9-56 来看看该页面底层的 HTML 代码。

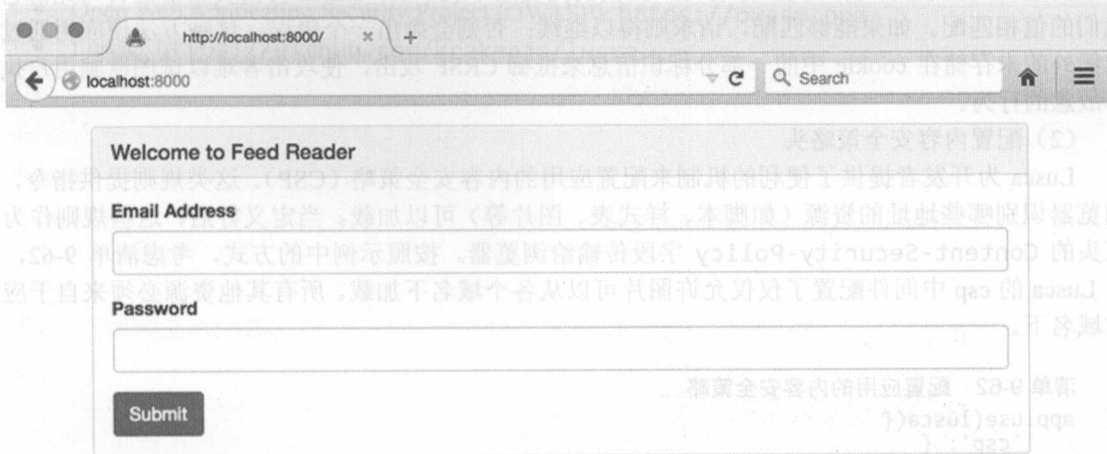
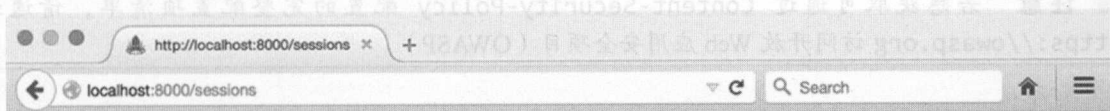


图 9-15 本章 app 项目的登录页面

在当前状态下，任何试图按这种方式登录的请求都会得到图 9-16 所示的错误。在此，我们看到来自于 Kraken 的错误信息警告：“缺少 CSRF 令牌”。



Internal server error

The URL /sessions had the following error Error: CSRF token missing.

图 9-16 Kraken 的“CSRF token missing”错误

该错误可以通过在应用的登录表单中添加一个单一的隐藏表单来解决。清单 9-61 显示了应用更新后 Dust 模板的代码片段，以及渲染输出后的片段。

清单 9-61 向登录表单中插入一个隐藏 csrf 字段

// app/public/templates/index.dust (摘录)

```
<form method="post" action="/sessions">
  <input type="hidden" name="_csrf" value="{_csrf}">
  <!-- ... -->
</form>
```

// Rendered output

```
<form method="post" action="/sessions">
  <input type="hidden" name="_csrf" value="OERRGi9AGNPEYnNWj8skkfL9f0JIWJp3uKK8g=">
  <!-- ... -->
</form>
```

这里我们用名字 `_csrf` 创建了一个隐藏表单，Lusca 会自动将其值用相同的名称传递到模板的上下文中。渲染在本例中的值为 `OERRGi9AGNPEYnNWj8skkfL9f0JIWJp3uKK8g=`，它是 Lusca 为我们生成的遗传随机散列（如“同步令牌”）。当我们提交这个表单后，Lusca 会核实其值是否与先前它给我们的值相匹配。如果能够匹配，请求则得以继续；否则会抛出一个错误。这种方法使应用通过要求额外的未存储在 cookie 中的一部分标识信息来抵御 CRSF 攻击，使攻击者难以试图欺骗用户执行非故意的行为。

（2）配置内容安全策略头

Lusca 为开发者提供了便利的机制来配置应用的内容安全策略（CSP）。这类规则提供指令，供浏览器识别哪些地址的资源（如脚本、样式表、图片等）可以加载。当定义好后，这些规则作为响应头的 `Content-Security-Policy` 字段传输给浏览器。按照示例中的方式，考虑清单 9-62，其中 Lusca 的 `csp` 中间件配置了仅仅允许图片可以从各个域名下加载。所有其他资源必须来自于应用的域名下。

清单 9-62 配置应用的内容安全策略

```
app.use(lusca({
  'csp': {
    'default-src': '\self\ ',
    'img-src': '*'
  }
}));
```

■ **注意** 若想获取可通过 `Content-Security-Policy` 配置的完整配置项清单，请通过 <https://owasp.org> 访问开放 Web 应用安全项目（OWASP）。

9.5 小结

Node 社区已经被所谓的“Unix 哲学”强烈影响了，它提倡（除了其他事情以外）创造小的、突出重点的模块，设计原则应为把一件事做好。这种方法使 Node 通过培养开源模块的大型生态系统，繁荣了开发者平台。PayPal 将该哲学体现在 Kraken 的模式上，使其不作为一个单一的、整体的框架，而更像是一个集合，对基于 Express 应用提供和扩展基础构件。通过这种方法，PayPal 努力为 Node 生态系统贡献了大量模块，从而让开发者受益——不论这些开发者是否选择把 Kraken 当作一个整体来使用。

9.6 相关资源

- Kraken: <http://krakenjs.com/>
- Confit: <https://github.com/krakenjs/confit>

- Meddleware: <https://github.com/krakenjs/meddleware>
- Enrouten: <https://github.com/krakenjs/express-enrouten>
- Dust.js: <http://www.dustjs.com>
- SuperAgent: <https://github.com/visionmedia/superagent>
- SuperTest: <https://github.com/visionmedia/supertest>
- Mocha: <http://mochajs.org>
- Open Web Application Security Project (OWASP): <https://owasp.org>
- Lusca: <https://github.com/krakenjs/lusca>

Mach

一个虽然忽略了某些不寻常的特性和改进但却秉承着一整套设计思想的系统，比一个虽然包含众多优点但却表现出各种孤立和不协调的系统的要好。

——Frederick P. Brooks

我们从不缺少 Node.js Web 服务，像 Express/Connect、Kraken 和 Sails 都是非常受欢迎的选择。Mach 也是该领域的一个相对比较年轻的项目，它的前身 Strata.js 曾经被大家追随了好几年。Mach 由前 Twitter 工程师 Michael Jackson 创造，它包含几个明显的设计原则。

- HTTP 请求被无缝地传递给了 JavaScript 函数。
- promise 化的接口使得 HTTP 响应能被异步式地延迟，HTTP 错误也能通过 promise 链传递（具体请参见第 14 章中关于 promise 是如何工作的内容）。
- 请求和响应同时能享受 Node.js 的流（stream）带来的优势，因此体积大的数据能支持分块发送和接收。
- 可编写化的中间件可以轻易地扩展 Mach 的核心功能。

选择使用哪个 Web 服务，广义上和选择任何库、框架或编程语言类似，都应该依据项目特定的使用场景决定。Mach 除了对于任何基于网页的应用能带来强大的功能之外，还可以充当 HTTP 客户端和代理，可以将请求转发至虚拟主机（类似于 Apache 和 Nginx），也可以改写 URL（类似于 Apache 的 mod_rewrite 模块）。Mach 虽然是个 Node.js 模块，但是它的一些特性允许它能同时在浏览器里使用，因此它的应用面会更广。

10.1 章节例子

这一章的章节代码里会包含很多可执行的代码例子。如果例子可以执行的话，会在代码清单里指出它们对应的文件名，如清单 10-1 所示。

清单 10-1 一个假想的例子

```
// example-000/no-such-file.js
console.log('this is not a real example');
```

本章节中的大部分例子都会启动一个 Node.js Web 服务。除非特别指出，不然就假定每个清单中提到的 JavaScript 文件都会用来启动一个服务。例如，清单 10-2 中的命令会执行 index.js 文件，然后

在 `example-001` 文件夹下启动一个 Mach Web 服务。

清单 10-2 启动一个示例 Web 服务

```
example-001$ node index.js
>> mach web server started on node 0.10.33
>> Listening on 0.0.0.0:8080, use CTRL+C to stop
```

10.2 安装

Mach 是一个 Node.js 模块，它可能会通过 Node.js 的包管理器 **NPM** 来安装。本章的例子也会采用 **Q** promise 库和一堆其他的 **NPM** 模块。所有都会以依赖形式列在示例代码的 `package.json` 文件中，所以只需简单地在示例代码文件夹下运行 `npm install`，然后 **NPM** 就会下载和安装各个模块：

```
code/mach$ npm install
```

10.3 Mach Web 服务

从表面上看来，Mach Web 服务和很多其他 Web 服务相似，它的灵感源自一些已经被证实有效的模式和设计，并且只对那些非常有必要的功能重新造轮子。对那些使用过其他面向 **REST** 的 Web 服务（如 **Express** (JavaScript)、**Sinatra** (Ruby)、**Nancy** (.NET) 等），在 Mach Web 服务里创建处理 HTTP 请求的路由应该是一个非常直观和熟悉的过程。

当引入 Mach 作为一个应用的依赖之后，Mach 应用栈通过调用 `mach.stack()` 来创建。这是伺候 HTTP 请求的第一步。清单 10-3 中把 Mach 应用栈赋给了 `app` 变量。

清单 10-3 创建应用栈

```
// example-001/index.js
'use strict';
var mach = require('mach');
// ... load other modules ...

// create a stack
var app = mach.stack();

// ...
```

每个 Mach 应用都被称作一个“栈”，这是因为 HTTP 连接会经过一层层中间件。这些中间件是一段一段可编写的程序，在请求传递给路由之前和在响应返回给客户端之前可以对它们进行一些操作。另外，中间件也可能产生其他一些对 Web 服务环境极为重要的副作用。

Mach 会自带一些常用的 Web 服务中间件，我们在下一部分会进行介绍。在清单 10-4 的例子中，Web 服务会使用一个 `mach.logger` 的中间件，当 Web 服务器接收到请求时会在终端输出 HTTP 诊断信息。

清单 10-4 给应用栈添加一个中间件

```
// example-001/index.js
'use strict';
var mach = require('mach');
```

```
// ... load other modules ...

// create a stack
var app = mach.stack();

// add some middleware
app.use(mach.logger);

// ...
```

路由简而言之就是与特定的 HTTP 方法和 URL 形式相配对的一个函数，这个函数在服务接收到请求时会对请求进行处理。路由一般会最后加到应用栈中，加在中间件的后面，这样中间件就能在路由处理之前进行请求信息的解析，在其之后对响应信息进行操作。

清单 10-5 给应用栈添加 HTTP 路由

```
// example-001/index.js

// add some routes

app.get('/book', function (conn) { /*...*/ });
app.get('/book/:id', function (conn) { /*...*/ });
app.delete('/book/:id', function (conn) { /*...*/ });
app.post('/book', function (conn) { /*...*/ });
app.put('/book/:id', function (conn) { /*...*/ });
app.get('/author', function (conn) { /*...*/ });
app.get('/library', function (conn) { /*...*/ });
app.get('/', function (conn) { /*...*/ });

// ...
```

当所有中间件和路由都被加到应用栈里之后，Web 服务就能开始侦听请求了。将应用栈传给 Mach.serve() 可以创建出一个 HTTP 侦听器，它会侦听来自默认 HTTP 模式、主机和端口，即 http://localhost:5000 的请求。

清单 10-6 在 8080 端口启动应用栈服务

```
// example-001/index.js

// serve the stack on a port
mach.serve(app, 8080);
// or mach.serve(app, {port: 8080});
```

如果需要设置更多的选项，可以以选项哈希映射的形式传给 Mach.serve()。这些选项的键值描述如表 10-1 所示。为了简便起见，本章中的例子会采用端口号的形式。

表 10-1 Mach 服务的选项

选项属性	描述
host	只侦听所有目标为该主机名的连接，默认没有限制
port	侦听所有目标为该端口的连接，默认为 5000

续表

选项属性	描述
Socket	侦听所有连到该 Unix 套接字的连接，有这个值时，host 和 port 将被忽略
quiet	当为 true 时将不输出启动和关闭服务时的信息，默认为 false
timeout	收到 SIGINT 或 SIGTERM 信号之后到强制终止连接之间的延时
key	SSL 连接 (HTTPS) 的私有密钥
cert	SSL 连接 (HTTPS) 的 X509 公有证书

10.3.1 HTTP 路由

Mach 的路由除了可以处理那些使用最常用的 HTTP 方法的请求，也可以处理一些不太常用的 HTTP 方法：

- GET
- POST
- PUT
- DELETE
- HEAD
- OPTIONS
- TRACE

清单 10-7 中的 HTTP GET 路由会查询假想的数据库里所有的书，然后把查到的记录通过 JSON 数组的形式传给客户端。

清单 10-7 对一个路由的详细说明

```
// example-001/index.js
app.get('/book', function (conn) {
  /*
   * 1. 路由会返回 promise 对象，Q 可以对回调驱动的数据库模块进行适配，
   * 因此结果（或错误）能通过 promise 链中传递。
   * makeNodeResolver() 方法会返回一个回调来“喂”给 deferred 对象。
   */
  var deferred = Q.defer();
  db.books.all(deferred.makeNodeResolver());
  /*
   * 2. 通过调用 promise.then() 方法，给 promise 链添加处理函数
   */
  return deferred.promise.then(function (books) {
    /*
     * 3. Connection.json() 方法会返回一个 promise 对象。
     * HTTP 状态码将会以响应头的方式传送，books 数组也会序列化 JSON 形式。
     */
    return conn.json(200, books);
  }, function (err) {
    /*
     * 4. 错误发生时，HTTP 500 错误会被返回给客户端，
     * 这个时候序列化的 JSON 响应中的内容就是错误信息对象了。
     */
    return conn.json(500, {error: err.message});
  });
});
```

和几乎其他任何路由一样，在这个路由中有几件事情会发生：

第一，一个 `deferred` 对象被创建出来，这个 `deferred` 的对象最终可以生成一个 `promise` 对象，并作为路由的返回结果（参阅第 14 章，获取更多关于 `promise` 如何工作的详细解释，特别是关于值和错误是如何在 `promise` 链上传递的相关解释）。在这个例子中，`Q` `promise` 库创建了 `deferred` 对象，然后又通过 `makeNodeResolver()` 创建了一个特殊的回调函数。这个回调函数会直接传给 `database.books.all()` 方法，并且会把返回的值或者错误传给 `promise` 链。

第二，两个处理函数被添加到 `deferred` 的 `promise` 对象上：一个用来接收书本数据并将它返回给客户端，另一个用来接收数据库获取数据失败时收到的错误。

第三，每个处理函数通过调用 `conn.json()` 方法，将它们各自接收的数据转变成一个 HTTP 响应，并会带上 HTTP 状态和数据负载。这个方法是一个语法糖，它会封装 `conn.send()` 方法（后面会详细说明），设置合适的 `Content-Type` 头字段，序列化 JSON 对象，返回一个 `promise` 对象并将它传到 `promise` 链上。当这个 `promise` 被解决（`resolve`）之后，实际的 HTTP 响应就会被发送。

在一个终端会话中，`curl` 命令能用来发起 HTTP GET 请求到 `/book` 路由去。响应体会包含序列化的 JSON 形式的书本数据：

```
example-001$ curl -X GET http://localhost:8080/book
[{"id":1,"title":"God Emperor of Dune","author":"Frank Herbert"... }]
```

在运行 Mach 服务的终端会话中，`mach.logger` 中间件会在标准输出中打印 GET `/book` 的请求信息：

```
example-001$ node index.js
>> mach web server started on node 0.12.0
>> Listening on :::8080, use CTRL+C to stop
::1 - - [17/Mar/2015 19:58:07] "GET /book HTTP/1.1" 200 - 0.002
```

1. URL 参数

URL 参数是指 URL 路径中代表应用数据的部分，如唯一标识符。我们常常能看到 REST 化的 URL 被写成类似 `<entity-type>/<entity-id>/<entity-particular>` 的形式。清单 10-8 中的代码定义了一个路由，它通过书的 ID 获取一本特定的书。可以看到参数：`id` 前面有个冒号前缀。一个路由可以有任意数量的参数，但是每个参数都必须有唯一的名字，并且必须是一个完整的 URL 片段。参数在路由里可以通过 `conn.params` 对象上的属性获取，每个属性都会对应 URL 参数中的名字，但是去掉了冒号前缀。所有属性值都会被 Mach 当作字符串来解析。因为 ID 在数据库里是数字形式，这个参数在数据库请求使用之前会用 `Number` 函数转换数据类型。

清单 10-8 只含有一个 URL 参数的 REST 化的路由

```
// example-001/index.js
app.get('/book/:id', function (conn) {
  var id = Number(conn.params.id);
  var deferred = Q.defer();
  db.book.findById(id, deferred.makeNodeResolver());
  return deferred.promise.then(function (book) {
    if (!book) {
      return conn.json(404);
    }
    return conn.json(200, book);
  }, function (err) {
    return conn.json(500, {error: err.message});
  });
});
```

和清单 10-8 中的通用 `/book` 路由不同，这个路由会搜索一个特定的实体。它可能在数据库里存在，也可能不存在。如果数据库操作成功，但是获取的 `book` 对象是 `null`，即那条 ID 记录不存在，路由就会以空的 HTTP 404 响应来标记解决。

2. 查询字符串和请求体

虽然 Mach 会自动解析 URL 参数，然后让它们在 `conn.params` 对象上可以被访问到，但是必须调用 `getParams()` 方法来解析查询字符串 (query string) 和请求体 (request body)。这是因为请求体是通过流传输的，解析默认并不会自动执行，而是由开发者决定是否以及何时解析（如果这个过程听起来很烦琐的样子，别太担心，因为有 `params` 中间件会自动化这个过程，后文中会提到）。

在清单 10-9 中，`/author` 路由会接受一个叫 `genre`（文体类型）的查询参数，然后就会返回一个写那种文体类型的作者的数据。`connection` 对象的 `getParams()` 方法会返回一个 `promise`，然后把解析好的 `params` 对象传给解决回调函数 (resolution callback)。`params` 对象中的每个属性的命名源自 URL 中的参数、查询字符串中或请求体。

清单 10-9 从查询字符串中抽取值

```
// example-001/index.js
app.get('/author', function (conn) {
  return conn.getParams().then(function (params) {
    var deferred = Q.defer();
    db.author.findByGenre(params.genre, deferred.makeNodeResolver());
    return deferred.promise.then(function (authors) {
      return conn.json(200, authors);
    }, function (err) {
      return conn.json(500, {error: err.message});
    });
  });
});
```

清单 10-10 中的 `curl` 指令发送文体类型参数为 `Horror` 的请求给服务器，返回的响应里有一条作者的记录，这个作者写的文体类型中有一个就是 `Horror`。

清单 10-10 用 `curl` 发送一个带有查询字符串的请求

```
example-001$ curl -X GET http://localhost:8080/author?genre=Horror
[{"id":6,"name":"Dan Simmons","website":"http://www.dansimmons.com/",
"genres":["Science Fiction","Fantasy","Literature","Horror"]}]
```

`getParams()` 方法有两个其他有用的特性。它可以接受一个对象参数，这个对象中的键 (key) 表示需要解析的参数，值 (value) 表示解析对应的参数时需要用的解析函数。在清单 10-11 中，当请求体解析时，那些没有在对象参数中被指定的请求体参数会被忽略。JavaScript 基本类型函数 `String`、`Number` 和 `Date` 都会把字符串解析成反序列化的对象。这样，当 `params` 对象传给解决回调函数时，每个属性都会是正确的类型。另外，方法也支持使用自定义的函数来反序列化私有数据格式的请求体参数。

当参数解析完之后，新的书本记录就会在数据库中被创建，然后被序列化并且在响应体中返回给客户端。

清单 10-11 从请求体中抽取值

```
// example-001/index.js
app.post('/book', function (conn) {
  return conn.getParams({
```

```

    title: String,
    author: String,
    seriesTitle: String,
    seriesPosition: Number,
    publisher: String,
    publicationDate: Date
  }).then(function (params) {
    var book = Book.fromParams(params);
    var deferred = Q.defer();
    db.book.save(book, deferred.makeNodeResolver());
    return deferred.promise.then(function (result) {
      return conn.json(result.isNew ? 201 : 200, book);
    }, function (err) {
      return conn.json(500, {error: err.message});
    });
  });
});

```

Mach 支持反序列化 URL-encoded、multipart 和 JSON 三种格式的请求体。对于其他格式，可以添加自定义的中间件，在请求体进入路由函数之前就对其进行反序列化，也可以通过 `conn.request.content` 获取未经处理的请求体流。

清单 10-12 展示了用 `curl` 命令分别以 URL-encoded 和 JSON 的方式来 POST 新的书本数据，同时分别展示了两个请求响应的结果。

清单 10-12 用 `curl` 命令发送 POST 请求体

```

example-001$ curl -X POST http://localhost:8080/book \
-H "Content-Type: application/x-www-form-urlencoded" \
-d "title=Leviathan%20Wakes&author=James%20S.A.%20Corey&publisher=Orbit&publication
Date=2011-06-15T05%3A00%3A00.000Z"
{"id":10,"title":"Leviathan Wakes","author":"James S.A. Corey","publisher":"Orbit"...}

example-001$ curl -X POST http://localhost:8080/book \
-H "Content-Type: application/json" \
-d @new-book.json
{"id":11,"title":"Ready Player One","author":"Ernest Cline","publisher":"Random House
NY"...}

```

当不同来源的参数（也就是 URL 参数、查询字符串和请求体参数三种）有相同的名字的时候，会应用下面的冲突解决机制。

1. URL 参数相比查询字符串和请求体参数总是有更高的优先级。
2. 查询字符串相比请求体参数有更高的优先级。
3. 请求体参数只有在没有遇到上述两种冲突的时候才会被使用。

3. 发送响应

到目前为止，路由只发送 JSON 响应，但是 Mach 可以推送任何有效的 HTTP 响应内容到客户端。

`Connection` 对象中最底层的响应方法是 `Connection.send()`。这个函数可以接受一个 HTTP 状态码和要发送的响应体，响应体可以是流（stream）、缓冲（buffer）或字符串。`Connection` 对象上的很多其他响应方法（如 `json()` 和 `html()`）只是包装了一下上述操作，加上了一些合适的头字段之后再调用 `send()`。

表 10-2 列出了 Mach 所有的响应方法，以及每个方法接受的内容的类型和每个方法使用的默认 HTTP 响应头字段值（如果有的话）。除了 `back()` 方法，其他方法都指定了一个 HTTP 状态码作为第一个参数，第二个参数则是响应体内容。虽然状态码是个可选的参数，但本章中的例子都会明确地设置状态码。

表 10-2 Mach 响应方法

方法	数据负载	响应头默认值
Connection.send()	流、缓冲或字符串	(无)
Connection.redirect()	地址	302 跳转 Location:
Connection.back()	地址	302 跳转 Location:
Connection.text()	文本字符串	Content-Type: text/plain
Connection.html()	HTML 字符串	Content-Type: text/html
Connection.json()	JSON 对象或字符串	Content-Type: application/json
Connection.file()	文件内容 (流、缓冲、字	如果可以从文件类型推断出合适的 MIME 类型, 则 Content-Type 的值就会被
	串或路径)	设置; 而 Content-Length 的值的设置会根据以下情况而定; 通过给 file()方法传入哈希映射选项对象的形式指定的文件大小, 或者当传入的是一个路径时, Mode.js 可以根据该路径取得文件从而确定文件的大小

`redirect()`和 `back()`方法不会发送响应体, 它们只会设置响应头中的 `Location` 字段来引导客户端跳转到另一页。`file()`方法既可以接受文件内容 (以流、缓冲或字符串的形式), 也可以接受一个文件路径。该方法会根据路径读取文件内容到文件流中, 然后传送给客户端。

也许 Web 服务发送给浏览器最常见的响应是 HTML 响应。HTML 网页现在很少再被存成一个完整的文件了。开发者将 HTML 分拆成若干可用的组件, 将标记语言和模板语言混在一起, 并且将数据和模板绑定, 从而构建有效的动态 HTML。

在清单 10-13 中, swig 模板库将两个 swig 模板分别编译成两个函数: `library()` (用来展现用户的书库) 和 `err500()` (用来展现服务器错误)。当路由处理进来的请求时, 它会从数据库里加载书本数据并且用 `library()`方法将数据和 `library.swig` 模板绑定起来。这样就能生成有效的 HTML 字符串, 然后就能作为响应体传给 `conn.html()`。如果这个过程发生了错误, `err500()`方法会采取跟上面类似的过程, 根据错误模板生成错误信息。

清单 10-13 发送 HTML 响应

```
// example-001/index.js
var swig = require('swig');
// ...

var library = swig.compileFile('./library.swig');
var err500 = swig.compileFile('./err500.swig');

app.get('/library', function (conn) {
  var deferred = Q.defer();
  db.book.all(deferred.makeNodeResolver());
  return deferred.promise.then(function (books) {
    return conn.html(200, library({books: books}));
  }, function (err) {
    return conn.html(500, err500({err: err.message}));
  });
});
```

在清单 10-13 中, 使用 `conn.html()`相对 `conn.send()`的优势只是在于前者更为便捷, 因为 `conn.html()`会自动设置头字段 `Content-Type: text/html`。类似地, `conn.text()`会设置 `text/plain`的内容类型。

对于 Mach 没有封装的内容类型, 头字段可以通过调用 `conn.send()`手动设置。例如, 清单 10-14 中的路由以 XML 的形式而非 HTML 的形式传送书库的数据, 这种情况下, 它就在响应头上显式地设置 `Content-Type: application/xml`后, 再将 `promise`对象返回。

清单 10-14 手动设置 Content-Type 头字段

```
// example-001/index.js
var xmlify = require('./xmlify');
// ...

app.get('/library.xml', function (conn) {
  var deferred = Q.defer();
  db.book.all(deferred.makeNodeResolver());
  conn.response.setHeader('Content-Type', 'application/xml');
  return deferred.promise.then(function (books) {
    return conn.send(200, xmlify('books', data));
  }, function (err) {
    return conn.send(500, xmlify('err', err.message));
  });
});
```

不是所有的响应都会返回内容。`conn.redirect()`方法会发送一个带要跳转到的 URL 地址的 Location 头字段给 HTTP 客户端，我们大概会在给定路由的内容不再可访问时使用这个方法。相比之下，`conn.back()`只是引导客户端返回当前页面的来源页面，可以指定一个备选的 URL 参数，以便在请求的 Referer 头字段是空的时候可以使用（例如，当用户直接在地址栏里输入 URL 时，Referer 头字段就是空的）。

清单 10-15 展示了一个简单的从应用根目录跳转到 /library 路由的例子。

清单 10-15 发送一个重定向响应

```
// example-001/index.js
// ...

app.get('/', function (conn) {
  return conn.redirect('/library');
});
```

10.3.2 建立连接

到目前为止，我们可以发现很明显 Connection 对象是与客户端之间所有通信的中心。它负责每个 HTTP 请求和响应的技术细节，并且给中间件和路由提供了交互和操控 HTTP 响应的途径。

一个 Connection 对象会载入若干属性并提供给中间件和路由：

- location
- request
- response

1. Location 对象

Connection.location 属性包含请求的目标 URL 的相关信息。表 10-3 展示了它所包含的属性和数据。

表 10-3 Connection 的 Location 对象的数据

Location 的属性	描述	例子
href	完整的地址	http://user:pass@webapp.com:8080/admin/dashbo ard.html#news?showWelcome=1
protocol	带上冒号的协议名	http:, https:
auth	URL 认证信息（如果有的话）	user:pass

续表

Location 的属性	描述	例子
Host	完整的主机地址，包含任何非标准的端口号（如不包含 80 和 443）	webapp.com:8080
hostname	主机名	webapp.com
port	主机端口	8080
pathname	不包含查询字符串的 URL 路径	/admin/dashboard.html#news
search	包含问号前缀的 URL 查询字符串	?showWelcome=1
querystring	不包含问号前缀的 URL 查询字符串	showWelcome=1
query	解析成对象的 URL 查询字符串	{showWelcome: 1}

如果你觉得 location 对象的 API 优点眼熟，那是因为 Mach 从现代浏览器的 `window.location` 对象获取了一点灵感。

Connection 对象把 Location 对象和头数据封装起来，提供了一部分很实用的属性，如表 10-4 所示。

表 10-4 Connection 对象上的 Location 属性

Connection 的属性	描述	例子
path	Location.pathname + Location.search	/admin/dashboard.html#news?showWelcome=1
auth	Authorization 头字段的值或者 Location.auth	user:pass
isSSL	如果 Location.protocol 是 "https:"，则为 true，否则就是 false	true

2. 请求和响应消息

Connection 对象暴露了 `request` 和 `response` 属性，它们都是 `Message` 类的实例，而 `Message` 是 Mach 的一个用来封装 HTTP 消息的内部类型。

(1) 消息的头字段

清单 10-14 中的例子解释了如何用 `conn.response.setHeader()` 设置一个响应消息的头字段。响应消息对象也暴露了一个 `addHeader()` 方法，它和 `Message.setHeader()` 有相同的功能，但是使用时要注意。如果是 `setHeader`，它会覆盖之前已经存在的同名键值对。如果是 `addHeader`，Mach 会认为它需要添加到原来已经存在的同名头字段下，从而会创建一个多值的头字段。

调用 `Message.getHeader()` 可以获取一个特定名字的头字段值，如果头信息中含有相应字段，则会返回对应的值。

头字段也可以完全通过 `Message.headers` 属性来操作。它可以获取和设置内部的头字段哈希映射，其中映射的键就表示头字段名（如 `Content-Type`），而映射的值就表示相应的头字段的值。

(2) 消息的 Cookie

发往服务器的 HTTP 请求和来自服务器的 HTTP 响应都会带上 cookie。这些 cookie 是一些键值对，请求时会在 `Cookie` 头字段里，响应时会在 `Set-Cookie` 头字段里。Mach 的消息对象会解析这些 cookie，并且把结果以一个对象哈希的形式暴露在 `Message.cookies` 属性上，而 `Message.getCookie()` 和 `Message.setCookie()` 就是这些属性对应的方法。

(3) 消息的内容

请求体和响应体会以流的形式存在于每个对应 `Message` 对象下的 `Message.content` 属性中。这些流可以通过管道（pipe）导入到其他转换流（transformation stream）中，也可以在每个 `Message`

对象中被完全替换。如果设置 `content` 属性值的时候设置成一个字符串而不是流，那么它会自动转变成一个流。

若干 `Message` 对象的方法提供了其他获取内容流的途径。`Message.bufferContent()` 方法会把流读取到内存的缓冲区中，然后以返回一个 `promise` 的形式输出结果。当 `promise` 被解决 (resolve) 之后，缓冲区数据就可以开始被调用代码使用了。该方法可以传一个可选的 `length` 值参数来限制被读取到缓冲区中的数据量。如果实际的缓冲区长度超过限制，`promise` 就会失败。这个方法在我们需要整体地处理请求体或响应体时会非常有用。如果 `Message` 对象被缓冲过了，则它的 `isBuffered` 属性就会被置成 `true`。

`Message.stringifyContent()` 则会以返回一个 `promise` 的形式，输出内容的字符串值。两个可选的参数，即长度和编码方式都可以指定，从而可以限制转换数据的长度和采用正确的编码方式来编码。和 `Message.bufferContent()` 相似，如果指定最大长度，同时字符串长度超过这个限制，那么 `promise` 就会失败。

`Connection.getParams()` 方法内部会调用 `Message.parseContent()` 方法，但是 `Message.parseContent()` 方法也可直接调用，比如有必要的话可能会在中间件中出现。它会根据媒体类型 (如 URL-encoded 类型)，对消息内容应用合适的解析器，然后将得到的结果用 `promise` 的形式输出。同时该方法也接受一个表示最大长度的参数。

10.3.3 公共的中间件

`Mach` 会自带很多中间件模块，它们都是封装了一些相当标准的 Web 服务的方法。虽然 Web 服务即使没有使用这些中间件也能正常运行，这些中间件全是可选的，在需要的时候可以使用。

本章节中的每一个例子都使用了 `mach.logger` 中间件。当 `Mach` Web 服务运行的时候，每个 HTTP 请求/响应记录都会在终端输出。清单 10-16 中，我们通过使用 `app.use()` 方法将这个中间件传给应用栈。

清单 10-16 `mach.logger` 中间件

```
// example-002/index.js

// add some middleware
app.use(mach.logger);

// add some routes...
```

从深层来说，中间件只是一些有特定的特征值的简单函数。这个概念在后面会详细说明，但是总体而言，`app.use()` 方法的第一个参数是中间件函数，第二个是可选的配置参数。

中间件被添加到 `Mach` 应用里的先后顺序是非常重要的，因为每个中间件都有可能更改请求和响应。有一些中间件，像 `Mach.file`，可能会完全阻止连接进入其他中间件或路由函数。

当 Web 服务收到一个请求后，它会以一种上游 (upstream) 的方式通过中间件。每个中间件会依次处理请求，然后传递给下一个中间件，直到这个链式过程被其中一个中间件终止，或者被一个路由处理，或者是因无法正确处理而产生一个错误。然而，当请求被处理之后，连接就会以下游 (downstream) 的方式再重新流经各个中间件，从而能让每个中间件都有机会处理一下响应。图 10-1 粗略地解释了在请求流程和响应流程中，中间件是如何被运行的。

当例子中有越来越多的中间件加入时，中间件的顺序所造成的影响就会越来越明显。

1. 内容类型中间件

`Mach.contentType` 和 `Mach.charset` 是两个设置 `Content-Type` 响应头的简单函数。当 `Content-Type` 响应头完全缺失或者未指定 `charset` 时，它们就会自动调整这个响应头。如果在路由里用 `Message.send()` 返回同种类型的内容（如 XML 数据）的话，这两个函数就会特别有用。我们不需要在每个路由里指定 `Content-Type` 头，而是可以在中间件里做全局的覆盖。在清单 10-17 里，上述两个中间件函数都被加入了应用栈中。

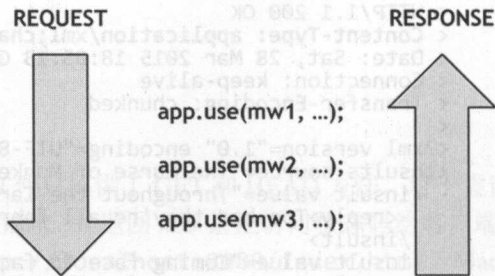


图 10-1 Mach 中间件处理请求和响应的顺序

清单 10-17 用 `Mach.contentType` 和 `Mach.charset` 设置默认的头字段值

```
// example-002/index.js
// ...

app.use(mach.charset);
app.use(mach.contentType, 'application/xml');

// ...
```

默认情况下，`Mach.charset` 使用 `utf-8` 编码，这在大部分情景下都够用了。如果要指定别的编码方式，可以通过给 `app.use()` 的第二个参数传个编码类型来实现。`Mach.contentType` 默认会用 `text/html`，但在这里指定了用 `application/xml` 这个值。

前面说过，添加到应用栈中的中间件的顺序是至关重要的。在这个例子中，`Mach.charset` 被加在 `Mach.contentType` 之前，这看起来似乎很匪夷所思，因为 `charset` 毕竟只是 `Content-Type` 头字段的一部分（从这个角度来说，首先应该设置的是头字段）。但是，我们回忆一下，响应是以上游的方法通过中间件的。因为内容类型和编码类型只有在路由返回内容给响应对象之后才能被确定，所以这两个中间件是以与它们添加时反向的顺序被调用的。

清单 10-18 中的 `curl` 命令请求了一个地址，这个地址的路由里会从磁盘上读取一个 XML 文件流，并且没有指定 `Content-Type` 头。从 `curl` 请求详细的输出里可以看出，Mach 中间件给这个请求的响应设置了默认的 `Content-Type` 头和默认的编码方式。

清单 10-18 自动设置 XML 内容的 `Content-Type` 头

```
// example-002/index.js
// ...
app.use(mach.charset);
app.use(mach.contentType, 'application/xml');

var insultsFilePath = path.join(__dirname, 'insults.xml');
app.get('/insults', function (conn) {
  conn.send(200, fs.createReadStream(insultsFilePath));
});
```

```
example-002$ curl -v -X GET http://localhost:8080/insults
* Hostname was not found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /insults HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:8080
```

```

> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/xml; charset=utf-8
< Date: Sat, 28 Mar 2015 18:05:13 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
<?xml version="1.0" encoding="UTF-8"?>
<insults source="The Curse of Monkey Island">
  <insult value="Throughout the Caribbean my great deeds are celebrated!">
    <reply>Too bad they're all fabricated.</reply>
  </insult>
  <insult value="Coming face to face with me must leave you petrified.">
    <reply>Is that your face? I thought it was your backside!</reply>
  </insult>
  <insult value="I can't tell which of my traits has you the most intimidated.">
    <reply>Your odor alone makes me aggravated, agitated, and infuriated!</reply>
  </insult>
</insults>

```

2. 静态文件中间件

Mach.file 中间件能用来伺候一些磁盘上物理文件夹下的静态文件（如.html、.css、.js 文件）。当请求过来时，**Mach.file** 会尝试将请求中的路径名和磁盘上的路径进行匹配。如果匹配成功，**Mach.file** 就会将静态文件内容以流的形式传给响应连接。如果没有匹配到结果，连接就会被传到下一个中间件（或路由）。

使用 **Mach.file** 中间件实际上就是把这个中间件函数添加到应用栈，同时指定静态文件所在的文件夹。在清单 10-19 中可以看到，一个哈希映射对象被传给了 **app.use()** 的第二个参数。这个对象里是一些用来配置 **Mach.file** 的选项，包括必须给出的 **root** 文件夹设置项。在这个例子中，**root** 文件夹指定了 **example-003/public**。

清单 10-19 Mach.file 中间件

```

// example-003/index.js
var path = require('path');
// ...

var publicDir = path.join(__dirname, 'public');
app.use(mach.file, {
  root: publicDir
  // ...other options...
});

// routes

```

提示 因为只有 **root** 是必须设置的选项，所以除了设置哈希映射对象外，可以直接以文件夹路径字符串的形式作为 **app.use()** 的第二个参数。

清单 10-20 展示了 **example-003/public** 文件夹下静态文件的文件树。**Mach** 把这个文件夹视为根路径，所以该文件夹下的文件和文件夹的 URL 路径都是相对于/的（如 **http://localhost:8080/styles/index.css**）。

清单 10-20 Public 文件夹下的内容

```

├── images
│   ├── bat-cat.jpg
│   └── computing.gif

```

```

├── llama.gif
├── no-idea.jpg
├── swine.gif
├── index.html
├── scripts
│   └── index.js
├── styles
│   └── index.css
└──

```

因为静态文件内容都是只读的，所以 Mach 对这些文件只提供了 GET 和 HEAD 方法。对于尝试请求在指定的静态文件夹之外的路径，Mach 会拒绝这些请求，并返回 403 禁止访问状态码给客户端。

当启动 Mach Web 服务之后，在浏览器里打开 `http://localhost:8080/index.html`。Mach 会展示静态页面 `index.html` 以及页面上的所有内容，如图 10-2 所示。



图 10-2 用 Mach.file 伺服静态 HTML 页面

你可能已发现，在 URL 中 `index.html` 被显式地包含进来。而通常的做法是将 `index.html`（或者其他等同的默认 `.html` 文件）映射到服务根目录路径，或者其他内嵌的文件夹目录路径。如果在上面的例子中将文件名从路径中去掉，那么 Mach.file 中间件就会返回一个 404 未找到的响应。为了改进这种情况，使得服务能自动伺服 `index` 文件，我们可以增加一个 `index` 属性到上面的哈希映射对象中。如果这个属性的值是 `true`，Mach.file 会自动在当前 URL 所表示的目录下（包括根目录）搜索 `index.html` 文件。如果想要获得更精细的控制，这个属性还可以是个数组。这个数组包含一连串文件名，Mach 会按文件名所在位置的先后顺序寻找对应的文件。清单 10-21 展示了这个属性以及它可能的值。

清单 10-21 通过 Mach.file 的 `index` 选项实现目录路径下自动搜索 `index` 文件

```
// example-003/index.js
// ...
```

```
app.use(mach.file, {
```

```
root: publicDir,
index: true
//or, index: ['index.html', 'index.htm', ...]
});
```

加上 `index` 选项后再重启 Web 服务，访问 `http://localhost:8080` 就会自动在浏览器展现 `index.html` 文件了。

`Mach.file` 还可以给没有 `index` 文件的文件夹目录自动索引文件。在清单 10-22 里可以看到，通过 `autoIndex` 就能开启这个功能。

清单 10-22 通过 `Mach.file` 的 `autoIndex` 选项自动索引没有 `index` 文件的文件夹目录

```
// example-003/index.js
// ...
```

```
app.use(mach.file, {
  root: publicDir,
  autoIndex: true
});
```

打开 `http://localhost:8080/images`，就可以看到页面列出了一个图片列表，它们的大小、MIME 类型以及上次修改时间都被详细列出，如图 10-3 所示。

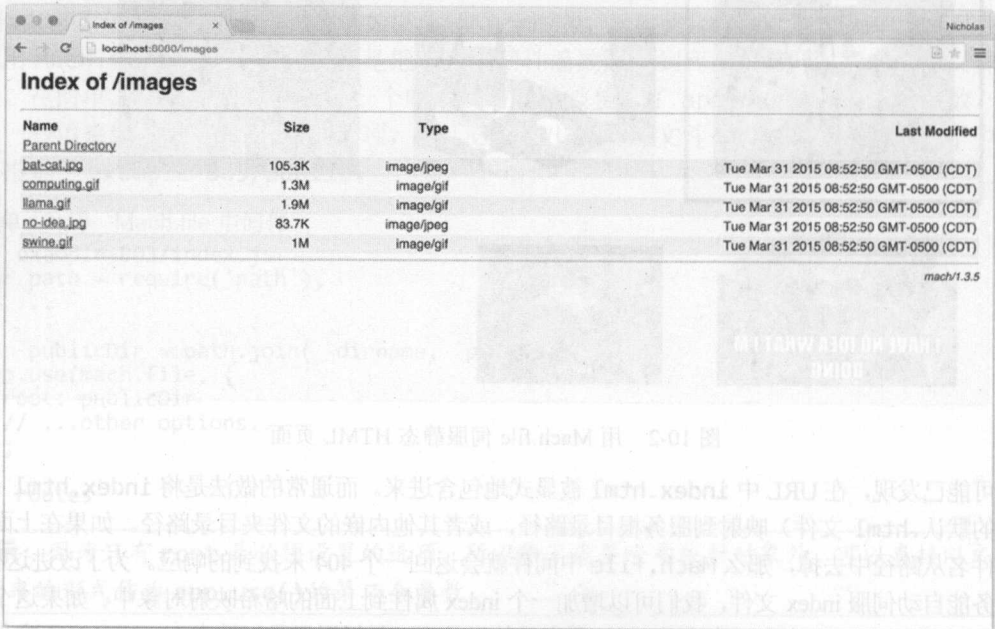


图 10-3 自动索引 `images` 文件夹

每张图片的名字是个指向自己的超链接，`Parent Directory` 是指向上一级的超链接，在这个例子中就是网站根目录。如果 `index` 和 `autoIndex` 同时使用了，`index` 有更高的优先级，`index` 指定的 `index` 页将会被展现，而不是自动索引文件。

3. 压缩中间件

现代浏览器会通过自动在每个请求里附上 `Accept-Encoding: gzip` 头，向服务器请求压缩后

的资源。压缩能显著减少响应的体积大小，提高响应速度，降低所需的带宽占用。而压缩的代价是服务器在压缩时和浏览器在解压时会增加一点点消耗。

Mach 的 `gzip` 中间件会自动压缩带有下列 `Content-Type` 头字段的响应。

- `text/*`
- `application/javascript`
- `application/json`

响应数据体会用 `Node.js` 的 `zlib` 模块来压缩，并且下列头字段会被加到响应头上。

- `Content-Encoding: gzip`
- `Content-Length: [compressed content length]`
- `Vary: Accept-Encoding`

提示 这个 `Vary` 头告诉任何网络中途的 HTTP 缓存设备：该响应需要根据 `Vary` 的值所表示的头字段的不同而保存多个缓存版本，在这个例子里就是 `Accept-Encoding`。如果对于同一个请求地址，请求 A 请求了一个未压缩的响应，而请求 B 请求的是压缩的响应，HTTP 缓存设备会同时缓存两个版本的响应，而不是只缓存一个。

清单 10-23 中，代码在静态文件中间件之前就加入了压缩中间件 `Mach.gzip`。当响应往上层返回经过 `Mach.gzip` 时，它会检查一下请求头中是否有 `Accept-Encoding: gzip`，然后看看 `Content-Type` 头是不是属于可压缩的范畴，如果两者都符合，那么响应体中的数据就会被压缩。清单 10-23 中的 `curl` 请求验证了上面这个逻辑。

清单 10-23 通过 `Mach.gzip` 压缩响应体

```
// example-004/index.js
// ...

app.use(mach.gzip);

var publicDir = path.join(__dirname, 'public');
app.use(mach.file, {
  root: publicDir,
  index: true
});

example-004$ curl -X GET -H "Accept-Encoding: gzip" -v http://localhost:8080/index.html
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /index.html HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:8080
> Accept: */*
> Accept-Encoding: gzip
>
< HTTP/1.1 200 OK
< Content-Type: text/html
< Last-Modified: Tue, 31 Mar 2015 13:52:50 GMT
< Content-Encoding: gzip
< Vary: Accept-Encoding
< Date: Tue, 31 Mar 2015 14:14:09 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
```

```

<
Htvu{y.
vvMvFy^
=[:RZ.6
Yt4XsF p1gg?#J\mYbvu@<g.xL
@t.t.:5|IHdI(
1l
qlyD$SR1$89q"Uon/V8q?TGq.
L.s^dd0#|L9E5dGtI

```

想要更精细地控制压缩算法（压缩等级、内存消耗、压缩策略等）的话，可以在 `Mach.gzip` 中间件添加的时候，再指定一个 `zlib` 的选项对象。至于每一个选项的技术细节，则超出本章的范围，具体可以参阅 Node.js 的 `zlib` 模块文档。

4. 数据体中间件

在之前的清单 10-11 中，`Connection.getParams()` 方法可以从查询字符串或 POST 请求体中解析和抽取数据。但是在每个路由里都用这个方法，很快会让人心生乏味。`Mach.params` 这个中间件就能使开发者免于这些重复枯燥的工作，它会自动地解析查询字符串和请求体中的数据，并把结果附在 `Connection.params` 对象上（URL 参数数据也在这个对象上），最后把连接传递给路由。

清单 10-24 中，当数据被 POST 到路由时，POST 数据体中的参数被附到 `conn.params` 对象上。然后这个对象作为一个数据库记录被保存到数据库中。`curl` 指令的输出结果表明 `Mach.params` 中间件的执行结果是我们所预期的。

清单 10-24 Mach.params 自动解析请求体参数

```

// example-005/index.js
// ...

// Mach.params
app.use(mach.params);

app.post('/hero', function (conn) {
  var deferred = Q.defer();
  db.hero.save(conn.params, deferred.makeNodeResolver());
  return deferred.promise.then(function (result) {
    return conn.json(201, result);
  }, function (err) {
    return conn.json(500, {err: err.message});
  });
});

example-005$ curl -X POST http://localhost:8080/hero \
-H "Content-Type: application/x-www-form-urlencoded" \
-d "name=Minsc&race=Human&class=Ranger&subclass=Berserker&alignment=Neutral%20
Good&companion=Boo"
{"id":6,"isNew":true}

```

提示 记住，URL 参数的优先级总是高于查询字符串参数和请求体的参数。如果不同来源之间发生了命名冲突，Mach 会最先考虑 URL 参数，然后是查询字符串参数，最后才是请求体参数。

为了确认之前 POST 的数据已经被添加到数据库，我们可以向清单 10-25 中的路由发起请求，并带上 `skip` 和 `take` 两个查询字符串参数。这两个参数使得客户端能对可能会比较大的英雄集合进

行分页，分页是通过指定一个偏移量（skip）表示从哪个地方开始，然后指定一个数量（take）表示该加载多个英雄的数据。因为 `Mach.params` 能同时处理请求体和查询字符串，所以没必要手动解析它们。

下面两个 `curl` 请求分别请求了第 1—3 条和第 4—6 条数据。刚才添加的英雄 `Minsc` 是最后一页的最后一个英雄。

清单 10-25 `Mach.params` 自动解析查询字符串参数

```
// example-005/index.js
// ...

// Mach.params
app.use(mach.params);
// ...

app.get('/hero'/*?skip=#&take=#*/, function (conn) {
  var skip = Number(conn.params.skip || 0),
      take = Number(conn.params.take || 0);
  var deferred = Q.defer();
  db.hero.page(skip, take, deferred.makeNodeResolver());
  return deferred.promise.then(function (heroes) {
    return conn.json(200, heroes);
  }, function (err) {
    return conn.json(500, {err: err.message});
  });
});

example-005$ curl -X GET http://localhost:8080/hero?skip=0&take=3
[{"id":1,"name":"Dynaheir"...}, {"id":2,"name":"Imoen"...}, {"id":3,"name":"Khalid"...}]

example-005$ curl -X GET http://localhost:8080/hero?skip=3&take=3
[{"id":4,"name":"Xan"...}, {"id":5,"name":"Edwin"...}, {"id":6,"name":"Minsc"...}]
```

5. 基本认证中间件

在 Web 应用中识别和追踪用户是一个很大的主题。`Mach` 提供了基本认证（basic authentication）来支持简单的安全应用场景，而持久化的会话（persistent session）可以支持更多的场景。

`Mach.basicAuth` 中间件可以像别的中间件一样添加到应用栈中，同时要求提供一个简单的验证函数作为唯一的参数。这个函数有两个参数：`username` 和 `password`。两者都是来自于对请求中的认证凭证解析后的结果。验证函数可能会返回下列三者之一的值：

- 被验证用户的用户名。
- 如果验证失败，返回 `false` 值。
- 一个 `promise` 对象，如果 `promise` 被解决，则会返回被验证用户的用户名；如果 `promise` 被拒绝，则返回 `false` 值。

清单 10-26 中的 Web 服务会给认证用户提供 `index.html` 文件。`Mach.basicAuth` 中间件会介入每一个请求，并去数据库里查询请求中提供的认证凭证。`db.user.byCredential()` 方法会返回一个 `promise`。如果认证通过（`resolve`），则会返回认证的用户；如果认证失败（`reject`），则会返回错误。如果认证通过，将返回用户名并传递到 `promise` 链上，最终会被用来设置 `Connection.remoteUser` 这个值。如果发生错误，将返回布尔值 `false`，并且会给客户端返回一个 401 未授权响应，并带上合适的 `WWW-Authenticate` 头字段值。

清单 10-26 使用基本认证来保证路由安全

```
// example-006/index.js
// ...

// Mach.basicAuth
app.use(mach.basicAuth, function (username, password) {
  return db.user.byCredential(username, password).then(function (user) {
    return user.username;
  }, function (/err*/) {
    return false;
  });
});

var indexPath = path.join(__dirname, 'index.html');

app.get('/', function (conn) {
  return conn.html(200, fs.createReadStream(indexPath));
});
```

当服务在运行时，如果一个用户尝试访问 `http://localhost:8080`，一个认证弹框会弹出让用户输入凭证。图 10-4 展示了 Chrome 浏览器中的认证弹框的样子。

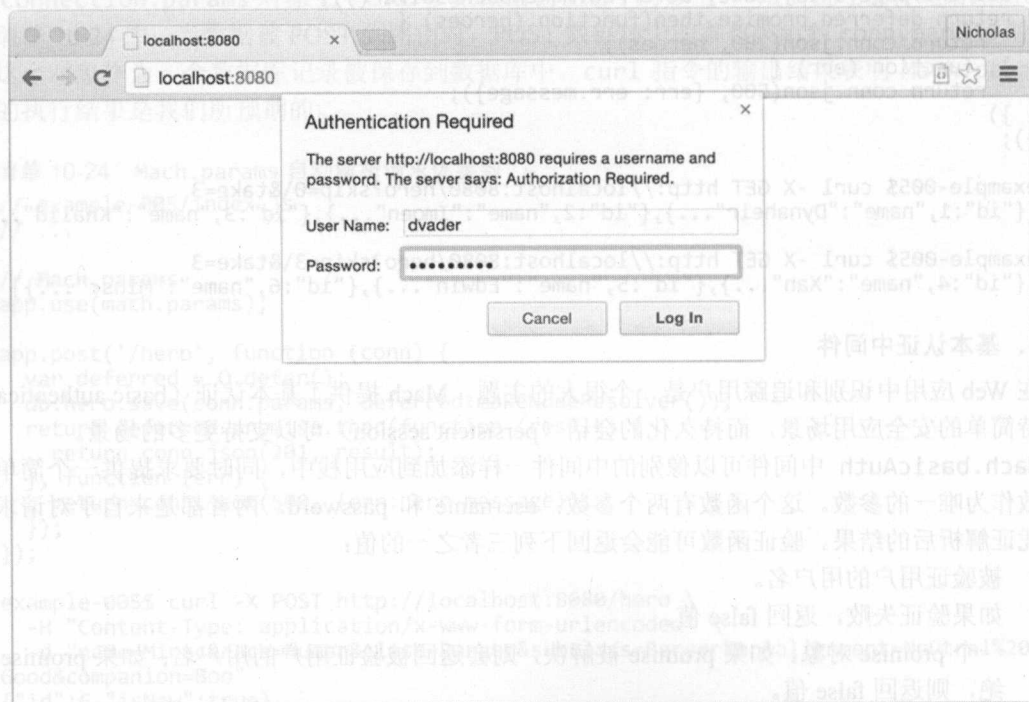


图 10-4 当基本认证失败时，浏览器弹出认证框让用户输入凭证

6. 会话中间件

当用户被认证成功之后，通常会用会话来追踪特定用户的数据。在应用栈里添加 `Mach.session` 中间件，就能自动启用会话 cookie 支持。`Mach.session` 选项对象里唯一必需的配置属性是会话密钥，它会被用来加密会话数据。清单 10-27 中，会话中间件被加在所有路由之前的位置。

清单 10-27 在应用栈中加入会话中间件

```
// example-007/index.js
// ...

var sessionSecret = 'c94ac0cf8f3b89bf9987d1901863f562592b477b450c26751a5d6964cbdc9eb085
c013d5bd48c7b4ea64a6300c2df97825b9c8b677c352a46d12b8cc5879554';

// Mach.session
app.use(mach.session, {
  secret: sessionSecret
});

var quizView = swig.compileFile('./quiz.swig');

app.get('/', function (conn) {
  return conn.html(200, quizView(conn.session));
});

// ...
```

清单 10-27 中的路由返回了一份 HTML 的测验题给浏览器（见清单 10-28）。这份测验题是一份 swig 模板，它会把会话对象里的 name、quest、colour 三个值插入模板中的三个 input 框中。当该路由第一次被访问到时，因为会话对象为空，所以这些 input 框都是没有值的。

清单 10-28 一份让人困惑的测验（你的答案是什么？）

```
<h1>Questions, three.</h1>
<form method="post" action="/questions/three">
  <fieldset>
    <h2>What... is your name?</h2>
    <div>
      <input name="name" type="text" value="{{name}}" />
    </div>
    <h2>What... is your quest?</h2>
    <div>
      <input name="quest" type="text" value="{{quest}}" />
    </div>
    <h2>What... is your favourite colour?</h2>
    <div>
      <input name="colour" type="text" value="{{colour}}" />
    </div>
    <div>
      <button>Cross the Bridge of Death</button>
    </div>
  </fieldset>
</form>
```

当 form 表单被 POST 到 /questions/three 路由（见清单 10-29），表单中的值从请求中抽取出来，这些值被用来填充会话对象。然后用户就被重定向到一个成功页，在这一页上可以让用户重新进行测验。

清单 10-29 在路由中设置会话的属性

```
// example-007/index.js
// ...

var successView = swig.compileFile('./success.swig');
var errView = swig.compileFile('./err.swig');

app.post('/questions/three', function (conn) {
```

```
return conn.getParams().then(function (params) {
  conn.session.name = params.name;
  conn.session.quest = params.quest;
  conn.session.colour = params.colour;
  return conn.html(201, successView());
}, function (err) {
  return conn.html(500, errView(err));
});
});
```

当用户返回测试页之后，表单中的每个问题会自动填充上次所填的答案。回忆一下清单 10-28，会话和测验模板是绑定在一起的。因为值之前保存在会话对象上，所以它们现在也可以再次被模板使用。图 10-5 展示了预填充好的表单值，以及用来连接浏览器和服务器端会话的会话 cookie。

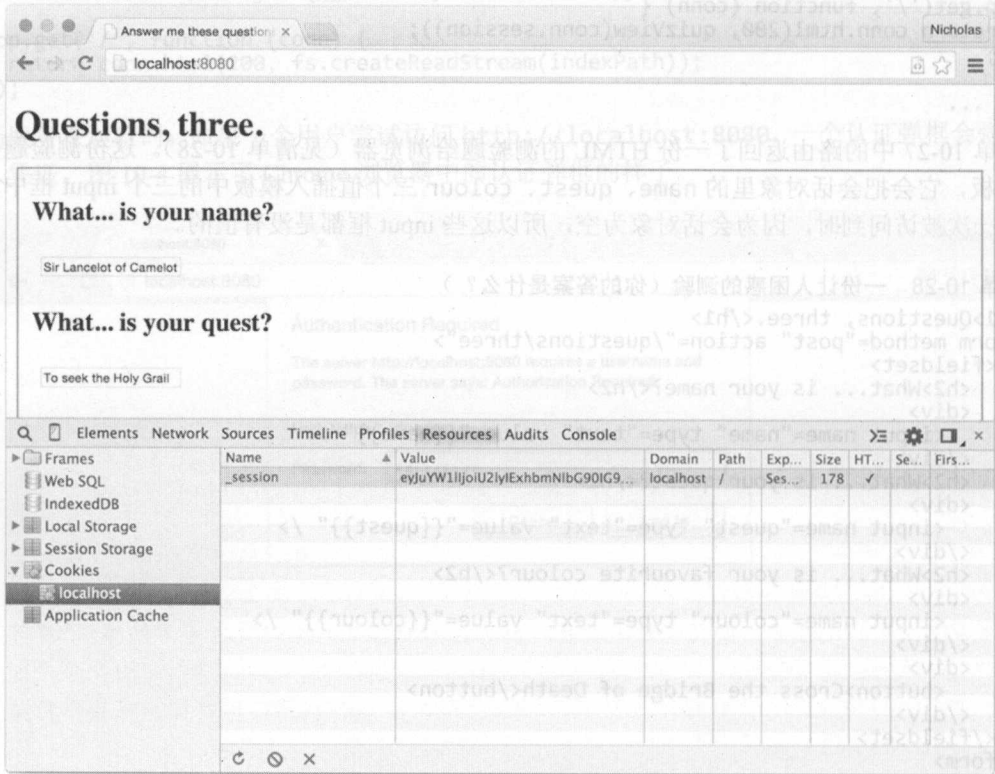


图 10-5 Mach 的会话 cookie

因为 `Mach.session` 默认使用 cookie 存储，所以在中间件添加到应用栈的时候，有很多额外的 cookie 专属的选项属性可以设置。这些属性列在表 10-5 中。

表 10-5 Mach 的会话 cookie 选项	
属性	描述
name	cookie 的名称，默认为 <code>_session</code>
path	cookie 的路径，默认为 <code>/</code>
domain	cookie 的域，默认为 <code>null</code>

续表

属性	描述
Secure	true 的话只能在 HTTPS 上传送 cookie, 默认为 false
expireAfter	cookie 过期还剩的秒数, 默认为 0 (从不过期)
httpOnly	true 的话限制 cookie 只能被 HTTP/S API 获取, 默认为 true

然而, Mach 的会话存储并不局限于 cookie。它支持内存会话和 Redis 会话。为了改变中间件的会话存储机制, 需要从 `mach/middleware/session/*` 中 `require()` 合适的模块。然后要设置选项对象下的 `store` 属性, 把 `store` 设置成那个被依赖的模块的实例。清单 10-30 展示了如何轻易地将默认的 cookie 会话存储替换成 Redis 会话存储。

清单 10-30 使用 Redis 作为会话存储

```
// example-008/index.js
// ...

var RedisStore = require('mach/lib/middleware/session/RedisStore');

// Mach.session
app.use(mach.session, {
  store: new RedisStore({url: 'redis://127.0.0.1:6379'})
});

// ...
```

7. Modified 中间件

当一个被请求的资源与上一次请求相比并未发生更改的时候, Mach 的 `modified` 中间件只要简单地通过标准的 HTTP 头就能告知 HTTP 客户端。Mach.modified 在发送响应之前, 能处理两种资源更改的情景。

Etag 和 If-None-Match

Web 服务能通过响应 Etag 头里包含一种版本标识 (通常是信息摘要), 标识一个特定版本的被请求资源。这个标识可以在后续请求相同资源时, 在请求的 `If-None-Match` 头上指定并发送回服务器。如果资源未发生更改, 也就是它的版本标识未发生改变, Web 服务就会返回 304 未更改响应, 并且在响应体中不会带上实际的资源数据。当这种情况发生时, 客户端就知道资源其实未发生改变, 所以它必须继续使用上次请求得到的数据。清单 10-31 展示了每个书本对象的摘要是如何被添加到每个书本路由的 Etag 响应头中的。

清单 10-31 在每个书本的响应中带上 Etag 头

```
// example-009/index.js
var jsonHash = require('./json-hash');
// ...

app.use(mach.modified);

app.get('/book/:id', function (conn) {
  var id = Number(conn.params.id);
  var deferred = Q.defer();
  db.book.findById(id, deferred.makeNodeResolver());
  return deferred.promise.then(function (book) {
    if (!book) {
      return conn.json(404);
    }
  });
});
```

```

    }
    conn.response.setHeader('ETag', jsonHash(book));
    return conn.json(200, book);
  }, function (err) {
    return conn.json(500, {error: err.message});
  });
});

app.put('/book/:id', function (conn) {
  var book = Book.fromParams(conn.params);
  var deferred = Q.defer();
  db.book.save(book, deferred.makeNodeResolver());
  return deferred.promise.then(function (result) {
    conn.response.setHeader('ETag', jsonHash(book));
    return conn.json(result.isNew ? 201 : 200, book);
  }, function (err) {
    return conn.json(500, {error: err.message});
  });
});
});

```

清单 10-32 中的第一个 curl 请求获取了一本 *Frank Herbert* 写的名叫 *Dune* 的书。响应中的 Etag 头显示这本书的信息摘要为 `cf0fdc372106caa588f794467a17e893`，响应体也包含序列化的 JSON 书本数据（Etag 信息摘要可能会因为操作系统的不同而不同。对每一个 curl 命令，使用 Etag 中的信息进行后续的对比）。

第二个 curl 请求使用了相同的 URL，但是带上了 `If-None-Match` 请求头，它的值是上次响应得到的 Etag 的值。因为服务器上这本书并没有发生变化（因而它的信息摘要会保持一致），Mach 会发送一个不带响应体的 304 未更改响应。

清单 10-32 使用 Etag 和 If-None-Match 头，测试内容更改

```
example-009$ curl -v -X GET http://localhost:8080/book/1
```

```

...
< HTTP/1.1 200 OK
< ETag: cf0fdc372106caa588f794467a17e893
< Content-Type: application/json
< Date: Mon, 06 Apr 2015 01:39:11 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
{"id":1,"title":"God Emperor of Dune","author":"Frank Herbert"...}

```

```
example-009$ curl -v -H "If-None-Match: cf0fdc372106caa588f794467a17e893" -X GET http://localhost:8080/book/1
```

```

...
< HTTP/1.1 304 Not Modified
< ETag: cf0fdc372106caa588f794467a17e893
< Content-Type: application/json
< Content-Length: 0
< Date: Mon, 06 Apr 2015 01:39:31 GMT
< Connection: keep-alive
<

```

清单 10-33 中，第一个 curl 请求是一个 HTTP PUT 请求，该请求把 *Dune* 的作者 *Frank Herbert* 的名字改为全名。第二个 curl 请求和清单 10-32 中的第二个请求一样，但是这次服务器响应的是 HTTP 200 OK，因为书本的信息摘要不一样了，表明书本资源发生了更新。后续的请求将会使用较新的信息摘要。

清单 10-33 更新后的 Etag 头通过了 If-None-Match 的校验

```
example-009$ curl -X PUT http://localhost:8080/book/1 \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "title=God%20Emperor%20of%20Dune&author=Franklin%20Patrick%20
  Herbert&publisher=Victor%20Gollancz&publicationDate=2003-03-13T06:00:00.000Z&series
  Title=Dune%20Chronicles&seriesPosition=4"
{"id":1,"title":"God Emperor of Dune","author":"Franklin Patrick Herbert"...}

example-009$ curl -v -H "If-None-Match: cf0fdc372106caa588f794467a17e893" -X GET http://
localhost:8080/book/1
...
< HTTP/1.1 200 OK
< ETag: 2595cd82c364b04473358bb2d0153774
< Content-Type: application/json
< Date: Mon, 06 Apr 2015 01:54:33 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
{"id":1,"title":"God Emperor of Dune","author":"Franklin Patrick Herbert"...}
```

Last-Modified 和 If-Modified-Since

Last-Modified 响应头和之前的部分提到过的 Etag 类似。但是跟 Etag 包含的是一个版本标识不同，Last-Modified 包含的是一个时间戳，它表示的是资源上次修改的时间。当 HTTP 客户端发送一个请求时，它可能会通过 If-Modified-Since 请求头提供一个时间戳，与服务器端该资源的实际时间戳进行对比。服务器只会提供更新版本的资源，否则，它只会返回一个 304 未更改的响应，从而告知客户端应该继续从缓存中使用之前未更改的资源，因为响应体中不会再包含未更改的资源了。

清单 10-34 中的代码使用每个作者记录里的 lastModified 时间戳，设置每个响应的 Last-Modified 头字段值。当作者记录更新时，数据库会自动更新这个记录的 lastModified 时间戳。

清单 10-34 在每个响应中增加 Last-Modified 头

```
// example-009/index.js

app.get('/author/:id', function (conn) {
  var id = Number(conn.params.id);
  var deferred = Q.defer();
  db.author.findById(id, deferred.makeNodeResolver());
  return deferred.promise.then(function (author) {
    if (!author) {
      return conn.json(404);
    }
    conn.response.setHeader('Last-Modified', author.lastModified);
    return conn.json(200, author);
  }, function (err) {
    return conn.json(500, {error: err.message});
  });
});

app.put('/author/:id', function (conn) {
  var author = Author.fromParams(conn.params);
  var deferred = Q.defer();
  db.author.save(author, deferred.makeNodeResolver());
  return deferred.promise.then(function (result) {
    conn.response.setHeader('Last-Modified', author.lastModified);
    return conn.json(result.isNew ? 201 : 200, author);
  }, function (err) {
    return conn.json(500, {error: err.message});
  });
});
```

在清单 10-35 中, 第一个 curl 请求获取了作者 *Hugh Howey* 的信息。从响应中可以得知, 这个记录上次修改的时间是 2015-04-06T00:26:30.744Z。在第二个 curl 请求中, 这个 ISO 日期字符串被用作 If-Modified-Since 头字段的值, 然后 Mach 据此返回了 304 未更改响应。

清单 10-35 使用 Last-Modified 和 If-Modified-Since 头, 校验内容更改

```
example-009$ curl -v -X GET http://localhost:8080/author/1
```

```
...
< HTTP/1.1 200 OK
< Last-Modified: 2015-04-06T00:26:30.744Z
< Content-Type: application/json
< Date: Mon, 06 Apr 2015 01:41:31 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
{
  "id":1,"name":"Hugh Howey","website":"http://www.hughhowey.com"...}

```

```
example-009$ curl -v -H "If-Modified-Since: 2015-04-06T00:26:30.744Z" -X GET
http://localhost:8080/author/1
```

```
...
< HTTP/1.1 304 Not Modified
< Last-Modified: 2015-04-06T00:26:30.744Z
< Content-Type: application/json
< Content-Length: 0
< Date: Mon, 06 Apr 2015 01:42:27 GMT
< Connection: keep-alive

```

很容易推测, 一旦记录被更新 (lastModified 日期也会发生变化), Mach 响应的响应头中就会包含更新的 JSON 数据和新的 Last-Modified 头字段。清单 10-36 中的两个 curl 请求展现了这个过程。

清单 10-36 更新后的 Last-Modified 通过了 If-Modified-Since 的校验

```
example-009$ curl -X PUT http://localhost:8080/author/1 \
-H "Content-Type: application/x-www-form-urlencoded" \
-d "name=Hugh%20C.%20Howey&website=http%3A%2F%2Fwww.hughhowey.com&genres=Science%20
Fiction%2CFantasy%2CShort%20Stories"
{"id":1,"name":"Hugh C. Howey","website":"http://www.hughhowey.com"...}
```

```
example-009$ curl -v -H "If-Modified-Since: 2015-04-06T00:26:30.744Z" -X GET http://
localhost:8080/author/1
```

```
...
< HTTP/1.1 200 OK
< Last-Modified: 2015-04-06T02:09:01.783Z
< Content-Type: application/json
< Date: Mon, 06 Apr 2015 02:09:09 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
{
  "id":1,"name":"Hugh C. Howey","website":"http://www.hughhowey.com"...}

```

10.3.4 路由重写

Mach 可以用 Mach.rewrite 中间件重写请求 URL。虽然它并不像 Apache 的 mod_rewrite 模块那样复杂, 但是简单又灵活的 Mach.rewrite 用来处理常用的应用场景足够用了。

当 Mach.rewrite 被加到应用栈时必须指定两个参数:

- 一个匹配来源请求 URL 的正则表达式对象（或一个字符串，它会被强制转化成正则表达式对象）；
- 一个路由的路径，请求会被静默地转发到这个路径上。

考虑这样一个应用场景：一个作者把他的博客从基于 PHP 的系统迁移到使用 Mach 的 Node.js 系统。因为搜索引擎已经索引了他以前的博客，所以那些 URL 将会永远地固定。通过用 `Mach.rewrite` 设置重写规则，他可以保证博客的那些旧 URL 仍然可以被访问到，同时又可以把旧地址映射到新的路由上。

清单 10-37 中的 `Mach.rewrite` 中间件使用复杂的正则表达式对象来建立起一个参数捕获规则组，这些捕获后的参数会充当新的博客地址中的一部分，它们分别是 `year`、`month`、`day`、`slug` 四个参数。正则表达式之后是一个字符串，它表示重写之后的 URL，其中的占位符是和上面捕获的四个参数按捕获位置一一对应的。实际上，`Mach.rewrite` 是用 `String.prototype.replace()` 方法往重写的 URL 里插入捕获出来的参数的。

清单 10-37 重写带参数的 URL

```
// example-010/index.js

var blogView = swig.compileFile(path.join(__dirname, 'blog.swig'));
var errView = swig.compileFile(path.join(__dirname, 'err.swig'));

app.use(
  mach.rewrite,
  // converts: /index.php/blog/2015-04-02/bacon-ipsum-dolor-amet
  new RegExp('\\index\\.php\\/blog\\/([\\d]{4})-([\\d]{2})-([\\d]{2})\\/([\\w\\-]+)'),
  // into: /blog/2015/04/02/bacon-ipsum-dolor-amet
  '/blog/$1/$2/$3/$4'
);

// :year=$1, :month=$2, :day=$3, :slug=$4
app.get('/blog/:year/:month/:day/:slug', function (conn) {
  var year = Number(conn.params.year || 0),
      month = Number(conn.params.month || 0),
      day = Number(conn.params.day || 0),
      slug = conn.params.slug || '';
  var deferred = Q.defer();
  db.posts.find(year, month, day, slug, deferred.makeNodeResolver());
  return deferred.promise.then(function (post) {
    if (post) {
      return conn.html(200, blogView({posts: [post]}));
    }
    return conn.html(404, errView({message: 'I haven\'t written that yet.'}));
  }, function (err) {
    return conn.html(500, errView(err));
  });
});
```

对于一个 HTTP 客户端来说，如图 10-6 所示的网页浏览器，或者搜索引擎机器人，重写后的 URL 仍然是完全有效的——虽然在内部它们会被更改成另外不同的东西。这与 HTTP 重定向和转发不同，因为这些情况下，客户端需要自己分析响应头，然后加载另一个网页。在当前这个例子中，重写的方式显然是更明智的方式。

清单 10-38 中的重写规则实现的功能和上面类似，但是与上面采用正则表达式来匹配请求的 URL 不同，这里采用了简单的字符串，因为它不需要捕获参数。要注意的是，`Mach.rewrite` 在把字符串转换成正则表达式时会自动转义字符串。如果你自己先转义了一下，那么会造成重复转义，从而导致匹配失败。

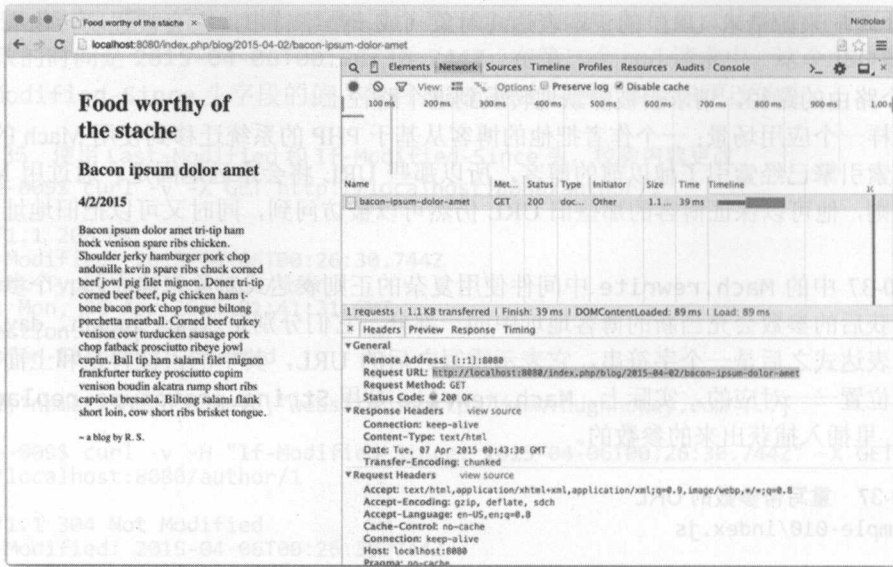


图 10-6 重写的 URL 在 HTTP 客户端里不会变化

清单 10-38 重写不带参数的 URL

```
// example-010/index.js
```

```
app.use(
  mach.rewrite,
  '/index.php/blog',
  '/blog'
);

app.get('/blog', function (conn) {
  var deferred = Q.defer();
  db.posts.all(deferred.makeNodeResolver());
  return deferred.promise.then(function (posts) {
    return conn.html(200, blogView({posts: posts}));
  }, function (err) {
    return conn.html(500, errView(err));
  });
});
```

10.3.5 主机映射

Mach.mapper 的特殊之处在于它在正常的路由机制上执行自己的路由方式。到目前为止，我们都假设路由路径都是存在于单个主机(localhost)下的，并且路径都是相对于主机名的。而 **Mach.mapper** 中间件则打破这一束缚，引入了一个中间件过滤器来实现同时根据主机名和 URL 路径名对请求进行路由。这有点像 Apache 的虚拟主机，只不过相比而言轻量许多。

为了证明 Mach 的映射功能是如何工作的，在清单 10-39 中，执行 **echo** 命令，往电脑的/etc/hosts 文件里加入两条别名。因为/etc/hosts 文件在类 Unix 系统上是受到保护的，所以用 **sudo** 命令提权。如果命令不成功，你也可以通过 **vim** 或 **nano** 编辑器手动往/etc/hosts 文件里添加别名。**cat** 命令可以把/etc/hosts 文件的内容输出到命令行里，所以您可以通过这种方式检验刚才所添加的别名记录。

清单 10-39 往/etc/hosts 里添加别名

```
example-011$ sudo echo "127.0.0.1 house-atreides.org" >> /etc/hosts
example-011$ sudo echo "127.0.0.1 house-harkonnen.org" >> /etc/hosts
example-011$ cat /etc/hosts
...
127.0.0.1 house-atreides.org
127.0.0.1 house-harkonnen.org
```

■ **提示** 如果你的电脑运行的是 Windows 系统，你就需要改 C:\Windows\System32\drivers\etc\hosts 了。因为这个文件被 Windows 系统保护，所以你需要用管理员权限运行编辑器来修改它。

一旦/etc/hosts 被修改后，用 ping 命令（见清单 10-40）来确认每个别名都指向了 127.0.0.1 (localhost)。

清单 10-40 用 ping 测试/etc/hosts 文件里的别名

```
example-011$ ping -t 3 house-atreides.org
PING house-atreides.org (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.044 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.118 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.074 ms
```

```
--- house-atreides.org ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.044/0.079/0.118/0.030 ms
```

清单 10-41 中的 Web 服务证明了 Mach.mapper 是如何工作的。它和任何正常的 Mach Web 服务一样：首先应用栈被创建，然后一些中间件被添加进去，接着就开始不一样了。两个额外并且独立的应用栈 `atreidesApp` 和 `harkonnenApp` 也同时被创建出来，而且各自被赋予了路由。实际上，所有的应用栈都有相同的路由，即 `GET/about`。

清单 10-41 Mach.mapper 中间件将应用映射到主机名

```
// example-011/index.js
// ...

var app = mach.stack();

app.use(mach.logger);
app.use(mach.params);
app.use(mach.file, path.join(__dirname, 'public'));

var atreidesApp = mach.stack();

atreidesApp.get('/about', function (conn) {
  var pagePath = path.join(__dirname, 'atreides.html');
  return conn.html(200, fs.createReadStream(pagePath));
});

var harkonnenApp = mach.stack();

harkonnenApp.get('/about', function (conn) {
  var pagePath = path.join(__dirname, 'harkonnen.html');
  return conn.html(200, fs.createReadStream(pagePath));
});

app.use(mach.mapper, {
  'http://house-atreides.org/': atreidesApp,
  'http://house-harkonnen.org/': harkonnenApp
});
```

```
app.get('/about', function (conn) {
  var pagePath = path.join(__dirname, 'about.html');
  return conn.html(200, fs.createReadStream(pagePath));
});
```

检查各个路由的内容后不难发现，每个路由函数当被调用时都会展现不同的 HTML 页面。这些路由能共存的原因在于，`Mach.mapper` 中间件在它的选项哈希对象中把 `atreidesApp` 应用栈映射到了 `hose-atreides.org` 主机名，将 `harkonnenApp` 映射到了 `house-harkonnen.org` 主机名。当请求被服务收到时，它们都会被传给 `Mach.mapper` 中间件。在那里，`Connection.hostname` 属性会被检查。如果它和选项对象中的键一样，连接就会传给对应的应用栈来处理。这里有几个有趣的结果：

- 主机名不同的应用栈可能会有相同的路由，如 `GET /about`。
- 因为中间件是直接附在应用栈上的，所以每个应用栈可能会有不同的中间件。
- 在 `Mach.mapper` 使用之前，添加到宿主应用栈的中间件将作用到所有 `Mach.mapper` 管理的应用栈上。
- 在 `Mach.mapper` 使用之前，添加到宿主应用栈的路由将先于基于主机名的路由执行。如果没有 `Mach.mapper`，主机名就不会被检查。要是宿主应用栈上有同名的 URL 路径，它就会被先被执行，而不管主机名。
- 在 `Mach.mapper` 使用之后，添加到宿主应用栈的任何路由都会被看作“失败”的路由。如果请求的主机名对应的应用栈都没办法处理请求，请求才会进入这些路由。

提示 当添加主机到 `Mach.mapper` 时，协议名会起作用，但是端口不会。所以我们可以放心地忽略端口。`Mach` 只会监听一个端口。主机名那个键应该总是以/结尾。

运行这个服务，然后打开浏览器，导航至 `http://localhost:8080/about`。这会打开图 10-7 中的页面，它是由宿主应用栈上定义的 `/about` 路由生成的。之所以是这个路由，是因为当前请求的主机名是 `localhost`，它并没有匹配任何 `Mach.mapper` 配置中指定的主机名。

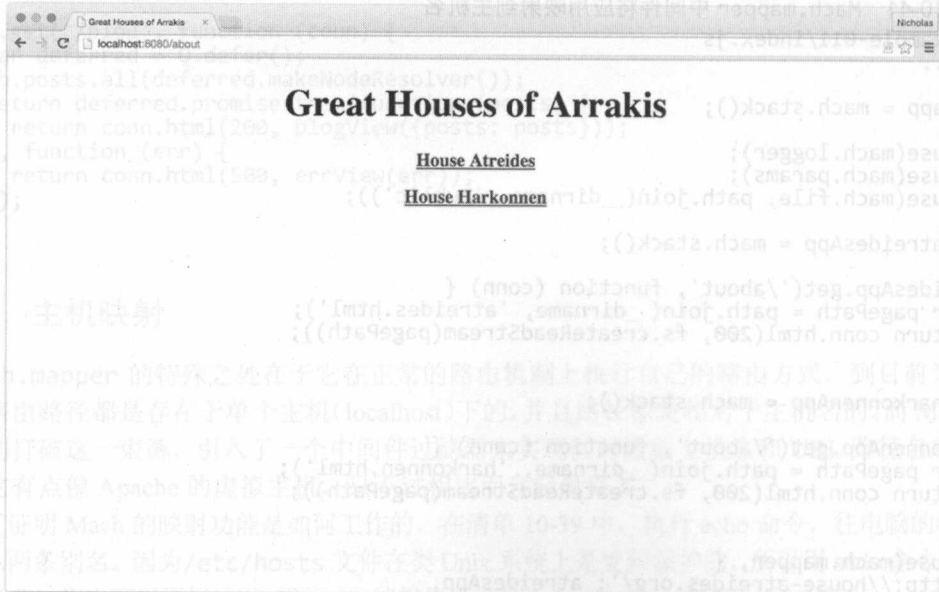


图 10-7 localhost 的/about 页

清单 10-42 中的页面源代码揭示了两个超链接 *House Atreides* 和 *House Harkonnen* 分别指向不同的主机。单击其中任何一个链接，就会展示 *Mach.mapper* 中定义的对路由所渲染的页面。要注意的是，虽然定义应用栈映射时端口号不重要，但是链接里必须明确地指出端口号，要不然浏览器就会自动理解成 80 端口。

清单 10-42 默认/about 页面中指向不同主机的链接

```
<h1>Great Houses of Arrakis</h1>
<h2>
  <a href="http://house-atreides.org:8080/about">House Atreides</a>
</h2>
<h2>
  <a href="http://house-harkonnen.org:8080/about">House Harkonnen</a>
</h2>
```

图 10-8 展示了 *House Atreides* 的 about 页，图 10-9 展示了 *House Harkonnen* 的 about 页。

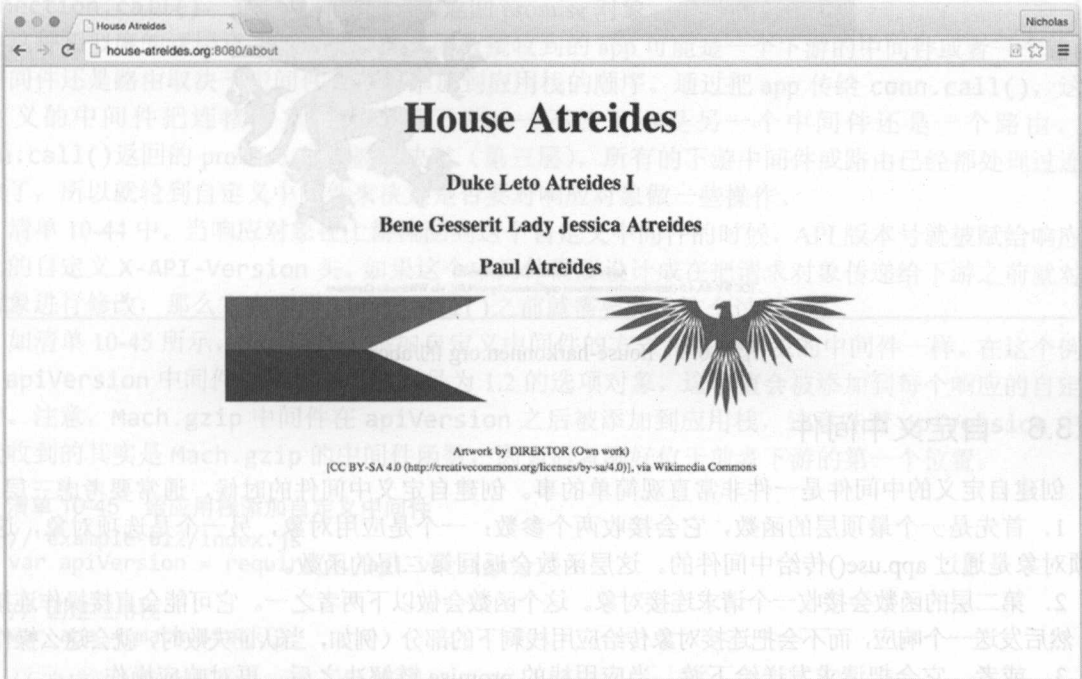


图 10-8 house-atreides.org 的/about 页

查看这两个页面的源代码，我们会发现有趣的事。两个页面中的图片，如清单 10-43 中的图片的 *src* 属性并没有指定主机前缀。

清单 10-43 图片地址不含主机前缀

```

```

这样做是可行的。因为 *Mach.file* 中间件会把 *example-011/public* 文件夹里的静态资源暴露出来，并且在 *Mach.mapper* 之前就被添加到宿主应用栈，所以它能影响到后面的所有应用栈。所有静态资源——图片、字体、脚本等——都可以存在相同的位置，并且不管是哪个主机名，所有

应用栈都能访问到。当然，有必要的话，每个应用栈下也可以用另一个 Mach.file 暴露不同的静态资源文件夹。

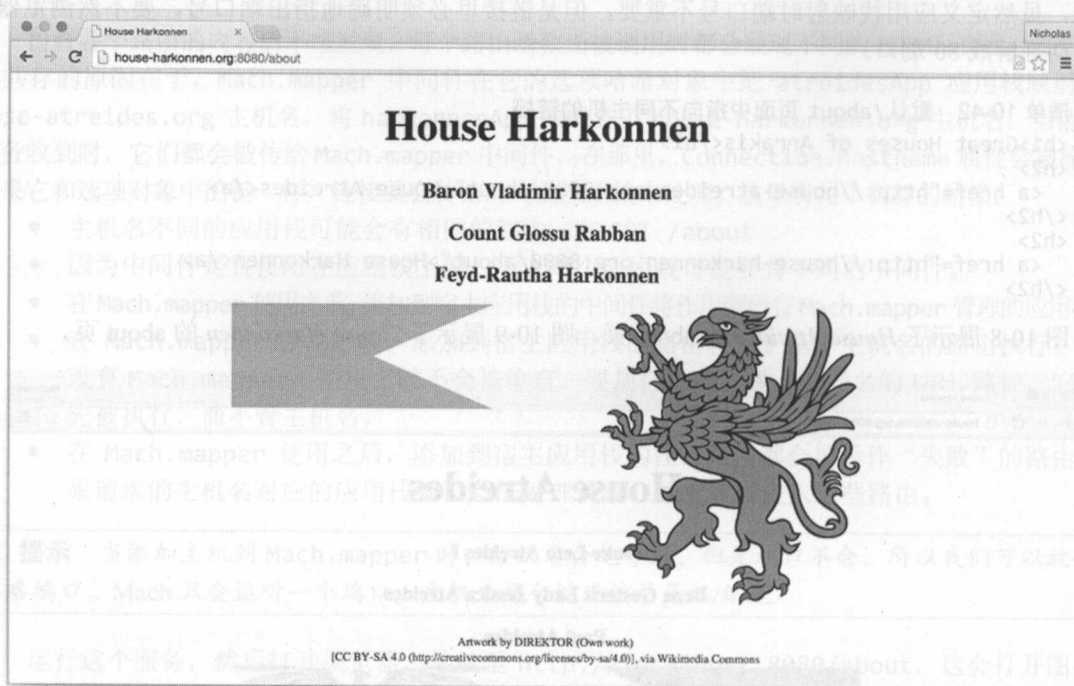


图 10-9 house-harkonnen.org 的/about 页

10.3.6 自定义中间件

创建自定义的中间件是一件非常直观简单的事。创建自定义中间件的时候，通常要考虑三层：

1. 首先是一个最顶层的函数，它会接收两个参数：一个是应用对象，另一个是选项对象，这个选项对象是通过 `app.use()` 传给中间件的。这层函数会返回第二层的函数。
2. 第二层的函数会接收一个请求连接对象。这个函数会做以下两者之一。它可能会直接操作连接对象，然后发送一个响应，而不会把连接对象传给应用栈剩下的部分（例如，当认证失败时，就会这么操作）。
3. 或者，它会把请求发送给下游，当应用栈的 `promise` 链解决之后，再对响应操作。

清单 10-44 中的中间件展示了上述所有的工作阶段。

清单 10-44 一个给响应添加应用版本头字段的自定义模块

```
// example-012/api-version.js
'use strict';

// 第一层
function apiVersion(app, options) {
  // 第二层
  return function (conn) {
    // 第三层
    return conn.call(app).then(function () {
      conn.response.headers['X-API-Version'] = options.version;
    });
  };
}
```



```

    });
  };
}

```

```
module.exports = apiVersion;
```

最顶层的 `appVersion()` 函数会通过 `module.exports` 暴露给外部。当中间件添加到应用栈时，这个函数会传给 `app.use()`。它会捕获应用实例和选项对象（第一层），把这两者固定在一个闭包之中，以供后续使用。当一个请求来了之后，被返回的函数（第二层）接收一个连接对象并做出决定。这个特殊的中间件唯一关心的就是在响应里加上 API 版本的头字段，所以在这个时候，它会调用 `Connection.call()` 方法，并把应用对象作为唯一的参数。

说到这里，你肯定会有不理解的地方。在 Mach 中，通过调用 `Mach.stack()` 方法创建的“应用栈”实际上是一个函数，它接收一个连接对象并返回 `Connection.call()` 的结果。这个过程中间件函数的处理过程是一样的。实际上，这个过程也和路由函数的处理过程近乎相等：它们都会调用 `Connection.call()`，并且所有路由都会返回 `promise` 对象。

这种相似度的深层意义在于，中间件函数接收到的 `app` 可能是一个下游的中间件或者一个路由，是中间件还是路由取决于中间件或路由添加到应用栈的顺序。通过把 `app` 传给 `conn.call()`，这个自定义的中间件把连接对象传递给了下游——不管下游是另一个中间件还是一个路由。当 `conn.call()` 返回的 `promise` 对象被解决时（第三层），所有的下游中间件或路由都已经都处理过连接对象了，所以就轮到自定义中间件来决定是否要对响应对象做一些操作。

清单 10-44 中，当响应对象往上游流回到这个自定义中间件的时候，API 版本号就被赋给响应对象上的自定义 `X-API-Version` 头。如果这个中间件需要设计成在把请求对象传递给下游之前就对请求对象进行修改，那么它在调用 `conn.call()` 之前就需要完成这个过程。

如清单 10-45 所示，往应用栈中添加自定义中间件的方式和添加原生的中间件一样。在这个例子中，`apiVersion` 中间件会接收一个版本号为 1.2 的选项对象，这个值会被添加到每个响应的自定义头中。注意，`Mach.gzip` 中间件在 `apiVersion` 之后被添加到应用栈，这意味着 `apiVersion` 中间件接收到的其实是 `Mach.gzip` 的中间件函数，因为后者刚好位于前者下游的第一个位置。

清单 10-45 给应用栈添加自定义中间件

```

// example-012/index.js
var apiVersion = require('./api-version');

// 创建应用栈
var app = mach.stack();

// 自定义中间件
app.use(apiVersion, {version: '1.2'});

// 原生中间件
app.use(mach.gzip);

app.get('/numbers', function (conn) {
  return conn.json(200, [4, 8, 15, 16, 23, 42]);
});

```

在清单 10-46 中，当请求发送到 Web 服务后，在响应里就能看到 `X-API-Version` 头。

清单 10-46 API Version 中间件的响应头

```

example-012$ curl -v -X GET http://localhost:8080/numbers
* Hostname was NOT found in DNS cache

```

```

* Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /numbers HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< X-API-Version: 1.2
< Date: Fri, 10 Apr 2015 01:41:42 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
[4,8,15,16,23,42]

```

10.4 MachHTTP 客户端

Mach 远不止是一个 HTTP 服务端。它的内部架构允许它在不同环境下能充当多种角色。仔细研读源代码之后就能发现，Mach 作为服务端的那部分功能实现实际上是一种扩展（extension）。Mach 的核心对象，如 **Connection**、**Location** 和 **Message**，可以在多种应用场景下使用。

清单 10-47 中的代码和前面的 Web 服务的例子很像。一个应用栈对象被创建出来，用来接收 HTTP 请求；文件中间件添加进来，用来伺服 `example-013/public` 下的静态内容；还有一个 `GET /mach/tags` 路由被注册到应用栈中。然而，这个路由中的代码利用 Mach 能充当 HTTP 客户端的特点发了一个 GET 请求，它会去向 Github API 请求所有 Mach 代码库的 tag。

清单 10-47 Mach 同时作为服务端和客户端

```

// example-013/index.js
var app = mach.stack();
app.use(mach.logger);
app.use(mach.file, {
  root: path.join(__dirname, 'public'),
  index: true
});

app.get('/releases', function (conn) {
  function addUserAgent(conn) {
    conn.request.setHeader('User-Agent', 'nicholascloud/mach');
  }
  return mach.get('https://api.github.com/repos/mjackson/mach/tags', addUserAgent);
}).then(function (conn) {
  var tags = [];
  JSON.parse(conn.responseText).forEach(function (tagData) {
    tags.push(tagData.name);
  });
  return tags.sort(semver.rcompare);
}).then(function (tags) {
  return conn.json(200, tags);
}, function (err) {
  return conn.json(500, {err: err.message});
});

```

Mach 的 HTTP 客户端方法和 Mach 的路由方法很像，但是它独立于应用栈而单独以一个 Mach 模块存在。Mach 可以发送任何标准方法的 HTTP 请求。

清单 10-47 中, `Mach.get()` 方法把请求 URL 作为第一个参数, 把一个函数作为第二个可选的参数, 这个函数可以在请求发送之前对其进行修改。这个请求会发送给 Github API, 并获取 `mjackson/mach` 代码库的所有 tag 信息。因为 Github API 要求所有请求都要有 `User-Agent` 头, 所以 `addUserAgent()` 方法是用来往请求中添加 `User-Agent` 头的。

和 Mach 的 API 其他部分类似, `Mach.get()` 方法会返回一个 `promise` 对象。如果 `promise` 解决了, 它的值会是带有响应信息的连接对象。如果被拒绝, 一个错误对象就会传给错误回调函数。

Github 的 JSON 数据会在 `Connection.responseText` 属性中以字符串形式存在 (或者在 `Connection.response.content` 中以流的形式存在)。一旦这个对象反序列化后, tag 名称就会被抽取出来, 倒序排列, 然后传给 `promise` 链。

在清单 10-48 中, 当服务器收到 `curl` 请求后, 所有 Mach 的发布 tag 都会以数组的形式列出来。

清单 10-48 用 `curl` 获取 Mach 的发布 tag 信息

```
example-013$ curl http://localhost:8080/releases
["v1.3.4","v1.3.3","v1.3.2","v1.3.1","v1.3.0" ...]
```

这里返回的 JSON 数据会被清单 10-49 中的 HTML 页面用到。注意到页面里也是用 `Mach.get()` 请求本地服务的。因为 Mach 适应不同环境的特征是作为一种扩展来实现的, 所以它不管在服务端代码还是客户端代码中都很有用。

■ **注意** 因为 Mach 是一个 Node.js 模块, 所以它可以被任何 CommonJS 模块加载器使用, 如 `Browserify` 或 `WebPack`。除这些情况外的情况, 如清单 10-49 中以普通脚本包含进来的这种情况, 都应该使用全局形式的 Mach 构建版本。

打开 `http://localhost:8080`, 浏览包含所有 Mach 发布版本的链接的列表。

清单 10-49 在浏览器中使用 `Mach.get()`

```
<!-- example-013/public/index.html -->
<h1>Mach Releases</h1>
<h2>Git you one!</h2>
<ul id="tags"></ul>
<script src="/vendor/mach.min.js"></script>
<script>
(function (mach, document) {
  var href = 'https://github.com/mjackson/mach/releases/tag/:tag';
  var ul = document.querySelector('#tags');

  mach.get('/releases').then(function (conn) {
    var tags = JSON.parse(conn.responseText);
    tags.forEach(function (tag) {
      var li = document.createElement('li');
      var a = document.createElement('a');
      a.innerHTML = tag;
      a.setAttribute('href', href.replace(':tag', tag));
      a.setAttribute('target', '_blank');
      li.appendChild(a);
      ul.appendChild(li);
    });
  });
})(window.mach, window.document))
</script>
```

10.5 MachHTTP 代理

虽然从技术上讲，Mach 的 HTTP 代理功能只是一个中间件，但是它本身可以用来创建一个完整的 HTTP 代理服务，或者接入已经存在的应用栈来代理特定的路由。这个代理功能在迁移 Web 应用的时候会非常有用。它能保证在一点点地迁移应用时还能把请求代理到酒店的 Web 应用上，或者，代理功能还能用来解决浏览器里的同源问题，它能将指向外部或第三方服务的请求借应用自身的代理来请求。

清单 10-50 中的代码创建了一个简单的 Mach 应用。它有一个根目录路由，同时能伺候 public 文件夹下的静态文件。在路由定义之后，代码通过调用 `Mach.proxy()` 创建了一个代理应用，代理指定了 HTTP 方法、主机名和端口。对这个例子来说，当 Web 应用运行时，它会监听端口 8080 上一部分请求，同时把另外部分的请求代理到另外一个监听着 8090 的 Web 服务。当 `Mach.proxy` 被传给 `app.use()` 时，这个代理应用栈会作为中间件的选项对象参数传入。

清单 10-50 将请求代理到另外一个服务上

```
// example-014/web.js
var app = mach.stack();
app.use(mach.logger);
app.use(mach.file, path.join(__dirname, 'public'));

app.get('/', function (conn) {
  var pagePath = path.join(__dirname, 'index.html');
  return conn.html(200, fs.createReadStream(pagePath));
});

var apiProxy = mach.createProxy('http://localhost:8090');
app.use(mach.proxy, apiProxy);
mach.serve(app, 8080);
```

通常中间件会在应用栈中被添加到路由之前，这样它们才有机会检查请求并在某些条件不满足的时候中断中间件 promise 链，或者修改请求，然后传递下去以供后续处理。不幸的是，`Mach.proxy` 是个相当无差别的中间件，它不会区别对待不同的请求，任何传给它的请求都会被它转发给代理服务。如果一个应用要混合使用本地路由和代理路由，有两种方法应对上面的“限制”。

- 将代理中间件加到路由的后面。这能保证如果应用路由能处理一个路由的时候，它就会处理它，同时阻止它继续往后传递给 `Mach.proxy`。这是清单 10-50 中使用的方法。
- 将代理中间件包裹在一个自定义中间件里，这个自定义中间件会区分对待不同的请求，然后只转发特定的需要转发的请求到代理服务去。因为它会过滤请求，所以它能被加到任何路由之前。这种方法在清单 10-51 中会阐述。

清单 10-51 将代理包裹在自定义中间件里

```
// example-014/web2.js
var apiProxy = mach.createProxy('http://localhost:8090');

app.use(function (app) {
  return function (conn) {
    if (conn.location.pathname.indexOf('/api') !== 0) {
```



```

    // not an API method, call the app stack normally
    return conn.call(app);
  }
  // API method, call the proxy app stack
  return conn.call(apiProxy);
});
});
});

```

```
app.get('/', function (conn) { /*...*/ });
```

毫无疑问，这个接受代理请求的模拟 API 服务是另一个 Mach 服务。它暴露了两个标准的 JSON 路由（见清单 10-52）：一个用来获取投票记录的列表，另一个用来投票。

清单 10-52 代理 API 服务的路由

```

// example-014/api.js
var votes = require('./votes');
// ...

app.get('/api/vote', function (conn) {
  var tallies = {};
  var voteCount = votes.length;
  votes.forEach(function (vote) {
    var tally = tallies[vote] || {
      count: 0,
      percent: 0.0
    };
    tally.count += 1;
    tally.percent = Number((tally.count / voteCount * 100).toFixed(2));
    tallies[vote] = tally;
  });
  return conn.json(200, tallies);
});

app.post('/api/vote', function (conn) {
  console.log(conn.params);
  var vote = conn.params.vote || '';
  if (!vote) {
    return conn.json(400, {err: 'Empty vote submitted.'});
  }
  votes.push(vote);
  return conn.json(201, {count: 1});
});

mach.serve(app, 8090);

```

■ **注意** 为了正常运行 example-014，web.js（或 web2.js）和 api.js 都需要用 Node.js 启动。Web 服务会监听 8080 端口的请求，API 服务会监听 8090 的请求。

Web 服务渲染了一个 HTML 页面作为一个小型投票应用的用户界面。虽然你知道并不是真正在选国王，但大家都是喜欢欢迎度测试的，而这个小应用能满足他们的测试欲望。图 10-10 展示了 <http://localhost:8080> 上渲染的页面。

当表单被提交后，事件处理函数会获得勾选项的值，然后把这个投票的数据通过请求发送给服务器。清单 10-53 中，sendVote() 方法发送了一个 AJAX POST 请求 /api/data 到 Web 服务上，然后这个请求被转发到代理 API 服务上。在那里，投票数据会被记录。

提交成功后，清单 10-53 中的 getTallies() 函数会发 GET 请求 Web 服务上的 /api/vote 来获取投票的数据。同样，这个请求会被转发到代理，然后返回 JSON 数据给客户端。

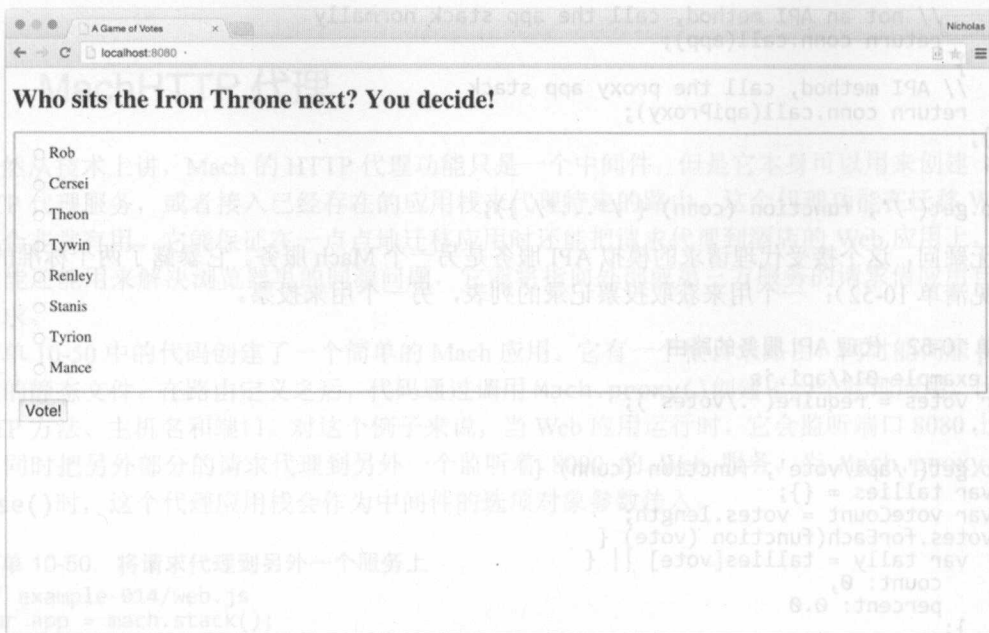


图 10-10 投票选国王

清单 10-53 提交投票表单

```
// example-014/index.html
var formPoll = document.querySelector('#poll');
// ...

function sendVote(vote) {
  function serializeVote(conn) {
    conn.request.setHeader('Content-Type', 'application/json');
    conn.request.content = JSON.stringify({
      vote: vote
    });
  }
  return mach.post('/api/vote', serializeVote);
}

function getTallies() {
  return mach.get('/api/vote').then(function (conn) {
    return JSON.parse(conn.responseText);
  });
}

formPoll.addEventListener('submit', function (e) {
  // ...
  var vote = checkbox.value;
  sendVote(vote).then(function () {
    // ...
    return getTallies().then(function (tallies) {
      // show tally data...
    });
  }).catch(function (error) {
    showError(error.err || error.message || 'The night is dark and full of errors.');
```

图 10-11 中网页展示的是来自代理的数据，数据由响应返回并被处理之后，展示在页面之上。

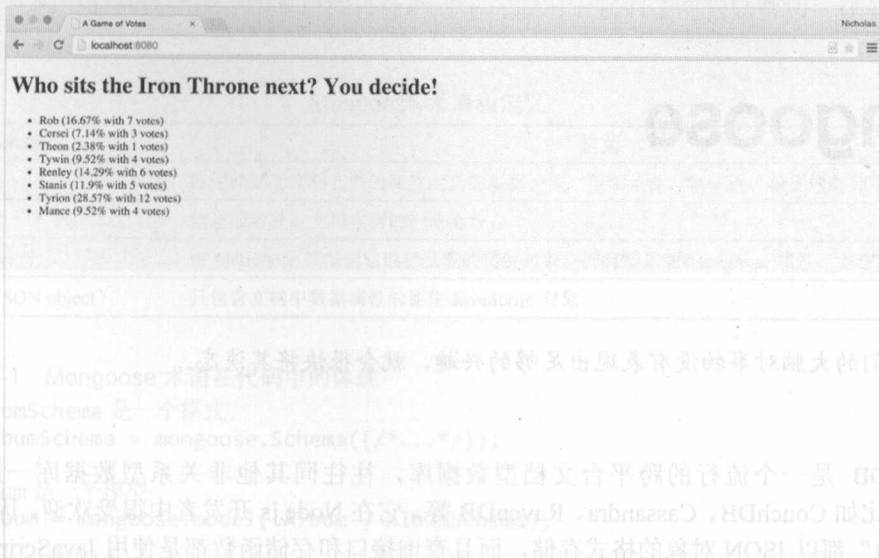


图 10-11 投票之后的结果

如果在代理请求的过程中发生错误，比如代理 API 服务器下线了，错误就会以 HTTP 错误的形式返回给客户端。因为这类错误是基础设施级别的错误，而不是应用级别的错误，比较明智地处理它们的方式是引入一个自定义的中间件来展现更易于用户理解的错误信息。

10.6 小结

虽然 Mach 并不是唯一的甚至也不是最流行的 Node.js Web 服务框架，但是它的极简的风格和简洁的 API 使得它非常灵活。它的核心架构保证它的常用组件能脱离环境在任何地方使用，同时能以扩展的形式加载那些在特定环境运行的组件。

自带的常用中间件集接入到基于 `promise` 的 API 后，能让请求链和响应链更加易于控制和操作。当需要更多功能时，自定义中间件创建起来也非常简便。

请求和响应信息是在 Node.js 原生的流的基础上构建的，请求查询和请求体数据都可以按需解析，给客户端的响应内容也会按数据块（`chunk`）返回。这种机制保证在进行 HTTP 操作时能有最低可能的内存占用和计算处理消耗。请求和响应内容也可以被转换和传输至缓冲区，然后可供各种格式处理器来解析，并可以转换成各种编码格式的字符串。

除了可以作为 HTTP 服务框架使用，Mach 也可以做其他一些重要的 HTTP 相关的工作：

- 重写请求 URL
- 将请求映射到虚拟主机
- 作为 HTTP 代理使用
- 发送 HTTP 客户端请求

总而言之，Mach 新颖的思想让它值得成为众多 Node.js 服务框架中的一员。

Mongoose

如果我们的大脑对事物没有表现出足够的兴趣，就会很快将其淡忘。

——西蒙斯

MongoDB 是一个流行的跨平台文档型数据库，往往同其他非关系型数据库一起被归类为“**NoSQL**”，比如 CouchDB、Cassandra、RavenDB 等。它在 Node.js 开发者中很受欢迎，因为它的“记录（records）”都以 JSON 对象的格式存储，而且查询接口和存储函数都是使用 JavaScript 实现的。

MongoDB 中的数据存储、访问和操作并不十分复杂，但像 Mongoose 这样的 Node.js 库能够帮助开发者将应用中定义的对象映射为 MongoDB 文档，而定义时涉及的模式、校验和行为并非 MongoDB 中（设计）的概念。Mongoose 为 MongoDB 实现了查询接口，开发者可以组合并链式地调用这些接口，从而大大简化了查询 API。

虽然 MongoDB 并非本章的直接主题，但在深入了解 Mongoose 之前，还是有必要先了解几个有关 MongoDB 如何工作的基本概念。如果你对 MongoDB 已经很熟悉，可以直接跳到下一节。

11.1 MongoDB 的基本概念

关系型数据库服务器存储了数据库模式（有时简称数据库），它包含相关的实体，比如表、视图、存储过程、函数等。数据库表包含元组（也被称为行或者记录）。一个元组由多个字段组成，每个字段都包含指定数据类型的值。元组是一维的，它的定义（数据类型和所包含的字段）由表决定。表中的所有元组尽管可能个别字段的值不同，但模式完全相同。元组的字段名和数据类型被称为元组的结构。

MongoDB 的数据分层模式表面上看起来很相似，如表 11-1 所示。

表 11-1 通过对比关系型数据库理解 MongoDB

RDBMS	MongoDB
服务器（Server）	服务器（Server）
模式（Schema）	数据库（Database）
表（Table）	集合（Collection）
元组（Tuple）	文档（Document）
字段（Field）	属性（Property）

表 11-2 定义了描述 Mongoose 组成的关键术语以及它们之间的关系。清单 11-1 中的代码演示了这些术语在代码中的具体体现。本章后续将详细讨论，但由于它们之间关联密切，在后续遇到困难时不妨参阅本节。

表 11-2 Mongoose 术语和定义

术语	定义
模式 (Schema)	应用级别为文档实例的属性定义的数据类型、限制条件、默认值、验证规则等
模型 (Model)	创建或者获取文档实例的构造函数
文档 (Document)	由 Mongoose 模型创建或者获取的实例对象，拥有特定的 Mongoose 属性、方法和数据属性
JSON 对象 (JSON object)	只包含文档中数据属性的原生 JavaScript 对象

清单 11-1 Mongoose 术语在代码中的体现

```
// albumSchema 是一个模式
var albumSchema = mongoose.Schema({/*...*/});

// Album 是一个模型
var Album = mongoose.model('Album', albumSchema);

// Album 是一个模型
Album.findById(/*...*/, function (err, album) {
  // album 是一个文档
  console.log(album);
});

// Album 是一个模型
Album.findById(/*...*/)
  .lean(true)
  .exec(function (err, album) {
    // album is a JSON object (because of `lean(true)`)
    console.log(album);
  });

// Album 是一个模型
Album.findById(/*...*/)
  .exec(function (err, album) {
    // album 是一个文档
    // toObject() 返回一个 JSON 对象
    console.log(album.toObject());
  });
```

不同于 RDBMS 元组，MongoDB 的文档不是一维的，而是可能包含其他对象和数组的完整 JSON 对象。实际上，相同集合 (collection) 中文档的属性无须相同，因为 MongoDB 的集合实际上是无模式的。在 MongoDB 的存储空间限制内，一个 MongoDB 的集合可以容纳任何形式 (模式) 和大小的文档对象。不过在实际使用中，尽管同一个集合中的文档可能包含一些可选的属性，或者可能包含的属性是任意数据，但所容纳的文档往往是类似“模式”的。在一般情况下，虽然 MongoDB 并不限制文档的模式，但应用自身通常会做一些特殊的限制。

默认情况下，MongoDB 的文档会自动分配一个名为 `_id` 的主键，主键类型为 MongoDB 特殊的 `ObjectId` 类型，并且会作为 MongoDB 集合的主索引。MongoDB 中也可以使用指定的字段作为主键。也可以使用附加字段为集合创建普通索引或者联合索引作为第二索引。

MongoDB 并不支持 RDBMS 中强大的外键概念。相反，MongoDB 依靠嵌套文档来存储关联数据。以 RDBMS 中经典的例子举例：客户、邮政地址和购物车订单。RDBMS 系统中可能会存在关联邮政地址和客户的外键（用于识别客户的居住地址），不过在 MongoDB 中把邮政地址作为嵌套对象存储在客户文档和订单文档中也能够满足需求，如清单 11-2 所示。

清单 11-2 MongoDB 中允许适当的数据冗余

```
// customer
{
  "_id": 1001,
  "name": "...",
  "postalAddress" {
    "street": "...",
    "city": "...",
    "state": "...",
    "zip": "..."
  }
}

// order
{
  "_id": 2001,
  "customer": 1001,
  "items": [
    { "sku": 3001, "qty": 2 }
  ],
  "shippingAddress" {
    "street": "...",
    "city": "...",
    "state": "...",
    "zip": "..."
  }
}
```

这种违背“参照完整性”（referential integrity）的做法可以从业务逻辑的角度加以解释。

- 订单或许从来不会发生变化。如果一张订单中有一些错误（比如送货地址填写有误），可以重新生成整个订单来替代错误的订单，并将正确的送货地址添加到新的订单上。
- 如果客户更改了邮寄地址，旧订单中的地址不会被更新，因此恰巧避免了数据完整性的问题。
- 或许更改邮寄地址始终只和客户相关联，而不应该影响订单。
- 客户（邮寄礼物时）可能会使用一个“临时”地址作为邮寄地址，而这种情况下不应当添加到客户的记录中。
- 统计不同的邮递数据应当从订单获取而非客户信息（比如，一个 C 级高管想要知道上个月有多少订单被发往密苏里，而并不关心这个月都有谁住在密苏里）。
- 磁盘空间或许很便宜，不强制遵循“参照完整性”并不会带来太多的损耗，相反，却可以获得不错的查询速度。虽然在 RDBMS 数据库中外键和引用完整性非常关键，但 MongoDB 面向文档的强大设计往往能够令问题迎刃而解。

最后是 MongoDB 的查询 API，尽管对于习惯写 SQL 的开发者来说可能会有点望而生畏，不过很快你会发现，查询数据涉及的大多数概念都相同：

选择 (find)、过滤 (where)、应用复合条件 (and, or, in)、聚合 (group)、分页 (skip, limit) 等。如何组合和执行查询的区别主要在语法上。

11.2 Mongoose 的一个简单示例

Mongoose 是针对 Node.js 应用的一个库。在使用 Mongoose 做开发(或者运行本章中的示例代码)之前, 需要首先在你的系统中安装 Node.js 和 MongoDB。使用默认的安装过程和配置, 足以运行本章中的示例代码。

■ 注意 阅读本章的前提是你已经对 Node.js 和模块有所了解, 而且知道如何通过 NPM 安装不同的模块。MongoDB 的实践经验将会对本章的理解大有帮助, 不过没有基础也无妨, 因为本章中的示例同 MongoDB 交互都将主要通过 Mongoose 实现。

部分示例会通过直接查询 MongoDB 来验证 Mongoose 的执行结果。本节将会演示 Mongoose 的一些基础概念, 这些概念在本章的后面也会详细讨论。这个示例涉及三个步骤:

1. 创建一个基础的 Mongoose 模式, 能够反映出 JSON 文件中的数据模式。
2. 读取 JSON 文件并通过 Mongoose 的模型将数据导入 MongoDB。
3. 运行一个基础的 Web 服务器, 使用 Mongoose 模型从 MongoDB 中获取数据并将其发送到浏览器。接下来的每个清单中第一行都会显示示例代码的文件路径。相应的示例文件将默认在命令行终端通过 Node.js 执行。

11.2.1 针对 JSON 数据创建一个 Mongoose 模式

Mongoose 文档代表应用级别的数据。在本章的示例应用程序, 音乐专辑的 JSON 文件定义了将被添加到 MongoDB 的初始数据集。清单 11-3 展示了 example-001/ albums.json 文件的模式: 一个专辑对象的数据。每个专辑对象都包含有关作曲家、标题、出版年份、曲目列表等信息。

清单 11-3 专辑的 JSON 数据文件

```
// example-001/albums.json
[
  {
    "composer": "Kerry Muzze",
    "title": "Renaissance",
    "price": 4.95,
    "releaseDate": "2014-01-13T06:00:00.000Z",
    "inPublication": true,
    "genre": ["Classical", "Trailer Music", "Soundtrack"],
    "tracks": [
      {
        "title": "The Looking Glass",
        "duration": {
          "m": 3,
          "s": 20
        }
      }
    ]
  }
]
// 更多 tracks...
```

```

    }
  } // 更多 albums...
}

```

Mongoose 是 MongoDB 的对象数据映射 (ODM)，所以 Mongoose 数据访问的核心是模型函数，用于查询所代表的 MongoDB 集合。一个 Mongoose 模型必须有一个名称便于引用，以及一个模式用于描述被访问和操作数据形式。清单 11-4 中的代码为专辑创建了一个模式，与 `example-001/albums.json` 中的 JSON 数据非常接近。模式在后面还会详细介绍，不过显而易见的是，模式为给定的 Mongoose 模型定义了一些属性及其数据类型。最后，通过模式创建了一个名为 “Album” 的模型函数。该模型函数在 `example-001/albums.json` 文件中被赋值给 `module.exports`，因此可以在 Node.js 应用的其他模块中引入。

提示 Mongoose 模式为模型定义了数据模式，模型函数为存储的文档数据提供了查询接口。模型必须有名称和模式。

清单 11-4 Mongoose 专辑模式和模型

```

// example-001/album-model.js
'use strict';
var mongoose = require('mongoose');

var albumSchema = mongoose.Schema({
  composer: String,
  title: String,
  price: Number,
  releaseDate: Date,
  inPublication: Boolean,
  genre: [String],
  tracks: [
    {
      title: String,
      duration: {
        m: Number,
        s: Number
      }
    }
  ]
});

var Album = mongoose.model('Album', albumSchema);

module.exports = Album;

```

11.2.2 使用 Mongoose 导入数据

既然专辑 (Album) 的模式和模型都已经定义好了，就可以通过 Node.js 脚本从 `albums.json` 文件中读取数据并使用定义好的专辑模型在 MongoDB 中创建文档。导入脚本需要执行三个步骤：

1. 使用 Mongoose 连接到正在运行的 MongoDB 服务器。
2. 读取并解析 `albums.json` 文件的内容。
3. 使用定义好的专辑模型 (Album) 在 MongoDB 中创建文档。

Mongoose 通过一个包含协议、主机和数据库的 URI 同 MongoDB 建立连接。清单 11-5 中的 URI 指向本地的 MongoDB 实例：`mongodb://localhost/music`。如果 MongoDB 实例中不存在该数据库，Mongoose 会主动创建，因此无须手动操作。如果 MongoDB 连接失败，Mongoose 将会抛出一个 `error` 事件；如果成功，则 Mongoose 会抛出一个 `open` 事件。清单 11-5 演示了如何通过回调函数对这两个事件进行处理。当 `open` 事件触发时，会读取 `albums.json` 文件并解析，之后会将专辑数组作为参数传递给 Album 模型的 `Album.create()` 方法并执行。这步操作会在 MongoDB 中创建专辑文档，稍后我们会通过 Album 模型查询这些数据。

清单 11-5 使用 Mongoose 导入专辑数据

```
// example-001/import-albums.js
'use strict';
var mongoose = require('mongoose');
var Album = require('./album-model');
var file2json = require('./file2json');
var fs = require('fs');
var path = require('path');

// 连接本地“music”数据库
// 如果数据库不存在，将自动创建数据库
mongoose.connect('mongodb://localhost/music');
var db = mongoose.connection;

db.on('error', function (err) {
  console.error(err);
  process.exit(1);
});

db.once('open', function importAlbums() {
  var albumsFile = path.join(__dirname, 'albums.json');
  file2json(albumsFile, 'utf8', function (err, albums) {
    if (err) {
      console.error(err);
      return process.exit(1);
    }

    console.log('creating %d albums', albums.length);

    // 使用模型大量地创建 albums
    // 如果数据集不存在，则自动创建数据集
    Album.create(albums, function (err) {
      if (err) {
        console.error(err);
        return process.exit(1);
      }
      process.exit(0);
    });
  });
});
```

在运行该脚本之前，首先需要在本地运行 MongoDB。有些 MongoDB 的安装过程会将 MongoDB 设置为自动启动，但有的安装过程会把决定权交给用户。要确定 MongoDB 是否已经运行，只需要在终端中执行 `mongo` 命令即可。如果 MongoDB 已经运行，你会看到类似清单 11-6 所示的输出。通过 `Ctrl+C` 快捷键，可以随时终止该进程。

清单 11-6 MongoDB 的终端客户端 mongo

```
$ mongo
MongoDB shell version: 2.6.7
connecting to: test
>
```

如果你看到的是一个报错信息，可以通过指定 MongoDB 的默认配置文件并执行 `mongod -f`，手动启动 MongoDB 服务。默认配置文件的位置因系统不同而有差异，因此你可能需要查阅 MongoDB 的安装文档。例如，假如在 OS X 系统中 MongoDB 是通过 Homebrew 安装的，配置文件的地址应该为 `/usr/local/etc/mongod.conf`。清单 11-7 展示了如何手动指定该配置文件的路径并手动启动服务。

清单 11-7 手动启动 mongod

```
$ mongod -f /usr/local/etc/mongod.conf
```

当 `mongod` 服务启动后，就可以使用 Node.js 运行 `example-001/import-albums.js` 脚本，将数据导入 MongoDB。清单 11-8 展示了执行脚本后的输出结果。

清单 11-8 运行导入脚本

```
example-001$ node import-albums.js
creating 3 albums
```

如清单 11-9 所示，当启动 `mongo` 终端客户端之后，接着执行一系列的命令（在 `>` 提示符后），验证 `music` 数据库和 `albums` 集合是否创建成功。执行 `show dbs` 命令，将显示当前 MongoDB 实例中所有的数据库。如果要查看一个数据库中的所有集合，需要首先使用 `use` 命令切换到数据库的环境，即你要查看的数据库名。接下来，执行 `show collections` 命令，查看当前数据库中的所有集合。在本例中，数据库 `music` 中有两个集合：`albums` 和 `system.indexes`（MongoDB 管理的一个集合）。

清单 11-9 验证专辑数据是否被成功添加到 MongoDB 中

```
$ mongo
MongoDB shell version: 2.6.7
connecting to: test
> show dbs
admin      (empty)
local      0.078GB
music      0.078GB
> use music
switched to db music
> show collections
albums
system.indexes
>
```

当选择 `music` 数据库后，可以通过一些基础查询来查看导入的专辑数据。在数据库的环境中，可以通过 `db` 对象访问到当前数据库的所有集合。集合作为 `db` 对象的属性，而且通过每个集合对象上的方法可以针对当前集合执行操作。例如，若要查看 `albums` 集合的记录数，可以针对该集合执行 `db.albums.count()` 方法，如清单 11-10 所示。同样，如果要查询专辑记录，可以执行 `db.albums.find()` 方法，通过指定标准查询参数 `query`（“where”语句）和 `projection` 参数（“select”语句）可以控制具体返回的数据。

清单 11-10 查询 albums 集合中的专辑数据

```
> db.albums.count()
3
```

```
> db.albums.find({}, {composer: 1})
{ "_id" : ObjectId("54c537ca46a13e0f4cebda82"), "composer" : "Kerry Muzzey" }
{ "_id" : ObjectId("54c537ca46a13e0f4cebda88"), "composer" : "Audiomachine" }
{ "_id" : ObjectId("54c537ca46a13e0f4cebdaa3"), "composer" : "Jessica Curry" }
```

在清单 11-10 的示例中，因为标准查询参数（传递给 `db.albums.find()` 的第一个对象）为空，所以会返回所有的结果集。不过，`projection` 对象指定了返回一个属性：`composer`。除了始终默认包含的 `_id` 属性之外的其他所有属性都会被排除，除非通过 `projection` 参数显式地指定排除 `id` 属性。

11.2.3 通过 Mongoose 查询数据

一旦专辑数据加载到 MongoDB 中，就可以使用和清单 11-4 中相同的模型来查询数据。

清单 11-11 所示代码使用 Node.js 的 `http` 模块来创建一个基础的 Web 服务，可以接收 HTTP 请求并返回 JSON 格式的响应数据。在这个例子中，为了简化，Web 服务针对所有 URL 请求的响应都是相同的。当接收到请求后，将通过 Mongoose 模型 `Album` 来查询 MongoDB 中的专辑文档。调用 `find()` 函数时传入了标准查询参数，`projection` 参数，以及一个回调函数。除了回调函数外，语法和清单 11-10 中查询专辑文档的 `db.albums.find()` 方法完全相同。

清单 11-11 使用 Mongoose 查询 MongoDB 数据库

```
// example-001/http-server.js
'use strict';
var mongoose = require('mongoose');
var Album = require('./album-model');
var http = require('http');
var url = require('url');

/*
 * HTTP 服务器将处理请求并响应
 */
var server = http.createServer(function (req, res) {
  Album.find({}, {composer: 1}, function (err, albums) {
    var statusCode = err ? 500 : 200;
    var payload = err ? err : albums;
    res.writeHead(statusCode, {'Content-Type': 'application/json'});
    res.write(JSON.stringify(payload, null, ' '));
    res.end();
  });
});

/*
 * 连接 MongoDB 实例，如果连接失败，将会报错
 */
mongoose.connect('mongodb://localhost/music');
var db = mongoose.connection;

db.on('error', function (err) {
  console.error(err);
  process.exit(1);
});

db.once('open', function () {
  /*
   * 开启 MongoDB 连接，并监听 HTTP 请求
   */
  server.listen(8080);
  console.log('listening on port 8080');
});
```

清单 11-12 运行 HTTP 服务器

```
example-001$ node http-server.js
listening on port 8080
```

在 Web 浏览器中访问 <http://localhost:8080>，或者如清单 11-13 所示，在命令行发送 curl 请求，都可以查看从 MongoDB 获取的专辑数据。

清单 11-13 向 HTTP 服务器发送一个 curl 请求

```
$ curl -v http://localhost:8080/
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Thu, 29 Jan 2015 01:20:09 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
[
  {
    "_id": "54c7020c342ee81670b261ef",
    "composer": "Kerry Muzzey"
  },
  {
    "_id": "54c7020c342ee81670b261f5",
    "composer": "Audiomachine"
  },
  {
    "_id": "54c7020c342ee81670b26210",
    "composer": "Jessica Curry"
  }
]
```

本章的剩余部分都将基于这个 Mongoose 模式、模型和存储在 MongoDB 数据库中的专辑数据。

11.3 使用模式 (Schemas)

Mongoose 模式是用于描述 MongoDB 文档中数据模式和数据类型的简单对象。由于 MongoDB 本身是无模式的，故 Mongoose 在应用层强制为文档定义确切的模式。模式定义的方式是通过调用 Mongoose 模块的 `Schema()` 函数并传递一个 Hash 对象，其中键名代表文档属性，值代表每个属性的数据类型。返回值是一个 Schema 类型的对象，并具备一些附加属性和辅助方法，可以扩展或者增强模式的定义。

11.3.1 数据类型

对于标量的属性，Mongoose 使用原生的 JavaScript 数据类型 `String`、`Boolean`、`Number` 和 `Date`，如清单 11-14 所示。

清单 11-14 Mongoose 模式中的原始类型

```
// example-001/album-model.js
var albumSchema = mongoose.Schema({
  composer: String,
  title: String,
  price: Number,
  releaseDate: Date,
  inPublication: Boolean
  // other properties...
});
```

对象字面量或者数组的属性值使用字面量定义每种类型 ({} 和 []). 嵌套对象字面量使用内联方式书写, 使用和自身属性相同的 Mongoose 模式。数组类型只包含一个元素, 用于定义该数组包含的对象类型。这里的类型可以是任何有效的 Mongoose 数据类型, 包括以内联方式定义的对象字面量, 类型作为数组的第一个元素。在清单 11-15 中, 音乐流派 **genre** 被定义为一个字符串数组, 而曲目 **tracks** 则被定义为一个对象字面量的数组。

清单 11-15 Mongoose 模式中的复杂类型

```
// example-001/album-model.js
var albumSchema = mongoose.Schema({
  // ...other properties
  genre: [String],
  tracks: [
    {
      title: String,
      duration: {
        m: Number,
        s: Number
      }
    }
  ]
});
```

Mongoose 自身提供了两种特殊的对象类型: **ObjectId** 和 **Mixed**。

当 MongoDB 中创建了一个文档后, 该文档会被分配一个 **_id** 属性作为这条记录的唯一标识。该属性使用 MongoDB 自身的 **ObjectId** 数据类型。Mongoose 中可以通过 **mongoose.Schema.Types.ObjectId** 引用该类型。这种类型很少直接使用。例如, 当根据 ID 查询文档时, 通常会使用该类型将字符串转换为要查询的 ID。

注意 当一个模式的属性包含任意数据 (请记住, MongoDB 是无模式的) 时, 可以通过 **mongoose.Schema.Types.Mixed** 类型定义。如果一个属性是 **Mixed** 类型, Mongoose 将不会追踪它的变化。Mongoose 保存文档的同时, 会创建一个内部查询, 只添加或者更新改变的属性。因为 **Mixed** 属性不被追踪, 所以应用程序必须在它改变的时候通知 Mongoose。通过 Mongoose 模型创建的文档会暴露一个 **markModified(path)** 方法, 用于强制 Mongoose 将 **path** 参数指定的属性标记为“已变更” (dirty)。

将一个 Mongoose 模式的属性设置为空的对象字面量 (没有任何属性) 后, Mongoose 会将其作为 **Mixed** 类型对待。

最后值得一提的是, Mongoose 是一个 Node.js 库, 它借助 Node 中 **Buffer** 类型的优势来存储大块的二进制数据, 比如图片、音频或者视频资源。因为二进制数据可能非常庞大, 许多应用将静态资

源存储在 CDN（内容分发网络），比如 Amazon 的简单存储服务（S3）。MongoDB 中不直接存储二进制数据，而是存储资源的 URL 引用地址。不同的应用，使用场景也不同。不过，Mongoose 的模式非常灵活，以上两种方式都支持。

11.3.2 嵌套模式

Mongoose 支持嵌套模式，即模式的属性类型可以是模式。这个特性非常有用，尤其是当大型模式需要共享公共的数据类型时，如客户和订单模式共享邮政地址的数据类型。清单 11-16 中的专辑曲目（track）的模式独立于专辑（album）的模式定义，而 `albumSchema.tracks` 属性又被赋值为曲目的模式。

清单 11-16 Mongoose 嵌套模式

```
// 将模式进行拆分
var trackSchema = mongoose.Schema({
  title: String,
  duration: {
    m: Number,
    s: Number
  }
});

var albumSchema = mongoose.Schema({
  // ...
  tracks: [trackSchema]
});
```

11.3.3 默认属性值

为模式属性添加有用的默认值，有助于 Mongoose 在创建文档时填写缺失的数据。这对于文档中一些非可选但具有已知值的属性来说是非常有用的。

在清单 11-17 中，专辑曲目模式中属性 `m` 和 `s`（minute 和 second）的默认值为 0。这是因为一首歌的长度完全有可能少于 1 分钟，或者是 X 分 0 秒。而专辑模式中 `releaseDate` 属性的默认值为 `Date.now` 函数。当某属性的默认值为函数时，Mongoose 会调用该函数，并将返回值转换为指定的属性类型后赋值给该属性。

清单 11-17 默认属性值

```
// 添加属性默认值
var trackSchema = mongoose.Schema({
  // ...
  duration: {
    m: {type: Number, default: 0},
    s: {type: Number, default: 0}
  }
});

var albumSchema = mongoose.Schema({
  // ...
  price: {type: Number, default: 0.0},
  releaseDate: {type: Date, default: Date.now},
  // ...
});
```

为属性添加默认值与为属性赋值不同。注意到 `m: Number` 已变成 `m: {type: Number, default: 0}`。通常，将某个属性赋值为对象会导致该属性变成 Mixed 类型或对象类型。但是，对象字面量的

属性类型则不会发生这种情况，Mongoose 会将对象字面量的属性值设置为键/值对。

11.3.4 必要属性

必要属性可应用于不可选属性的类型定义上。当文档保存时，文档模式所需的属性如有缺失，则会将返回的校验错误信息传递给保存方法的回调函数。在清单 11-18 中，专辑作曲家、专辑标题、曲目标题和曲目时长等对象都是必要属性。

清单 11-18 必要属性

```
// adding required attributes
var trackSchema = mongoose.Schema({
  title: {type: String, required: true},
  duration: {
    required: true,
    type: {
      m: {type: Number, default: 0},
      s: {type: Number, default: 0}
    }
  }
});

var albumSchema = mongoose.Schema({
  composer: {type: String, required: true},
  title: {type: String, required: true},
  // ...
});
```

如果在必要属性的定义中用字符串类型代替布尔类型，该字符串将被用作校验返回的错误消息。（快速执行文档的校验。）

清单 11-19 自定义必要属性的报错信息

```
var trackSchema = mongoose.Schema({
  title: {type: String, required: 'Missing track title!'},
  // ...
});
```

11.3.5 第二索引

Mongoose 文档保存到 MongoDB 时，会自动使用 `_id` 作为索引。

也可以为 Mongoose 模式添加第二索引，以提高对其他字段的查询性能。MongoDB 支持普通索引（单字段索引）和联合索引（多字段索引）。在清单 11-20 中，曲目和专辑模式添加了以下索引。

- 曲目标题（普通索引）
- 专辑作曲家（普通索引）
- 专辑标题（普通索引）
- 专辑标题+专辑作曲家（联合索引）
- 专辑类型（普通索引）

清单 11-20 为模式添加第二索引

```
// adding secondary indexes...
var trackSchema = mongoose.Schema({
```

```

    title: {type: String, required: true, index: true},
    // ...
  });

var albumSchema = mongoose.Schema({
  composer: {type: String, required: true, index: true},
  title: {type: String, required: true, index: true},
  // ...
  genre: {type: [String], index: true},
  // ...
});

albumSchema.index({composer: 1, title: 1});

```

在属性类型声明后添加一个索引字段，即可设置普通索引。而联合索引则需要通过 `Schema.index()` 方法进行设置。传递给 `index()` 方法的对象包括相应的模式中属性的属性名和一个数值，该数值可取值为 1 或 -1。

MongoDB 可通过索引进行降序或升序。联合索引使用数值类型确定每个字段的顺序，而不是布尔类型。对于普通索引来说，字段顺序并不重要，因为 MongoDB 可以通过任何字段进行搜索。但是对于联合索引来说，字段的定义顺序非常重要，因为字段顺序限制了 MongoDB 在使用联合索引查询时的排序操作。MongoDB 文档支持深度联合索引策略。

在清单 11-20 中，在专辑模式中同时添加了 `composer` 和 `title` 的普通索引和联合索引。用户可以分别通过 `composer`、`title` 搜索专辑，或是同时使用 `composer` 和 `title` 进行搜索。

11.3.6 模式校验

当保存 Mongoose 文档时，将进行模式规则校验。校验规则是为模式中的特定属性所定义的函数。该函数计算属性的值，并返回一个布尔值来指示校验的有效性。清单 11-21 显示了如何为模式对象添加属性校验。

清单 11-21 模式属性校验

```

// 添加模式校验
var trackSchema = mongoose.Schema({/*...*/});

var albumSchema = mongoose.Schema({
  // ...
  tracks: [trackSchema]
});

albumSchema.path('tracks').validate(function (tracks) {
  return tracks.length > 0;
}, 'Album has no tracks.');
```

模式的 `path()` 方法返回 `SchemaType` 的一个实例，该实例对象封装了模式属性的定义。在这个例子中，属性 `tracks` 的类型是专辑曲目对象数组。`SchemaType.validate()` 方法关联了 `track` 属性的校验函数，第一个参数就是校验函数。校验函数的唯一所需参数就是校验的值。`validate()` 方法的第二个参数是当校验不通过时返回的错误信息。当保存 `album` 文档时，校验函数将作为 Mongoose 校验过程的一部分被执行，该函数用于核定 `tracks` 属性，从而保证专辑中至少有一个曲目。也可以将校验规则作为模式属性的一部分进行定义。在清单 11-22 中，`tracks` 属性的定义中就包含了校验规则。`tracks` 的 `validate` 属性值是两个元素的数组（元组），该数组的元素 0 是校验函数，元素 1 是

错误信息。

清单 11-22 内联属性校验声明

```
function validateTrackLength (tracks) {
  return tracks.length > 0;
}

var albumSchema = mongoose.Schema({
  // ...
  tracks: {
    type: [trackSchema],
    validate: [validateTrackLength, 'Album has no tracks.']
  }
});
```

虽然 Mongoose 校验过程是异步的，但是校验函数却是同步的，如清单 11-22 所示。在大多数场景下，同步校验是完全可以接受的，但是在另一些场景下，需要使用异步校验。异步校验函数接收一个回调函数作为第二个参数。当异步校验完成时，调用该回调函数。并且在校验成功或失败时，将 true 或 false 传给回调函数。清单 11-23 显示了如何定义专辑 tracks 属性的异步校验函数。

清单 11-23 属性异步校验

```
albumSchema.path('tracks').validate(function (tracks, respond) {
  process.nextTick(function () {
    respond(tracks.length > 0);
  });
}, 'Album has no tracks.');
```

为了查看校验函数的执行结果，可以删除 example-002/albums.json 中每个专辑的 tracks 数据，删除之后的 JSON 数据，如清单 11-24 所示。

清单 11-24 曲目为空的专辑

```
// example-002/albums.json
[
  {
    "composer": "Kerry Muzzey",
    "title": "Renaissance",
    "price": 4.95,
    "releaseDate": "2014-01-13T06:00:00.000Z",
    "inPublication": true,
    "genre": ["Classical", "Trailer Music", "Soundtrack"],
    "tracks": []
  },
  {
    "composer": "Audiomachine",
    "title": "Tree of Life",
    "price": 9.49,
    "releaseDate": "2013-07-16T05:00:00.000Z",
    "inPublication": true,
    "genre": ["Classical", "Trailer Music"],
    "tracks": []
  },
  {
    "composer": "Jessica Curry",
    "title": "Dear Esther",
    "price": 6.99,
    "releaseDate": "2012-02-14T06:00:00.000Z",
    "inPublication": true,
    "tracks": []
  }
]
```

```

    "genre": ["Classical", "Video Game Soundtrack"],
    "tracks": []
  }
]

```

当保存文档时进行校验，也就是说，一旦调用 `Model.create()` 方法或 `save()` 方法，校验函数就会执行。如果校验不通过，错误信息就会传递给校验的回调函数。（稍后将详细讨论文档。）

如果再次运行导入数据程序，调用在 `example-002/import-albums.js` 中的 `Album.create()` 方法，使用以上删除 `tracks` 后的 JSON 数据来创建新的 Mongoose 文档。清单 11-25 显示了控制台输出的 `ValidationError` 序列，`ValidationError` 显示了 `tracks` 属性的校验失败消息。

清单 11-25 当模式校验失败时的控制台输出

```

example-002$ node import-albums.js
creating 3 albums
{ [ValidationError: Validation failed]
  message: 'Validation failed',
  name: 'ValidationError',
  errors:
    { tracks:
      { [ValidatorError: Album has no tracks.]
        message: 'Album has no tracks.',
        name: 'ValidatorError',
        path: 'tracks',
        type: 'user defined',
        value: [] } } }

```

将 `album` 模式和 `track` 模式分开，并分别添加默认属性值、必要属性、辅助索引以及校验函数之后，与 `example-001` 中的简单模式对比，新的 `album` 模式发生了很大的变化。清单 11-26 显示了这个新的更健壮的 `album` 模式。

清单 11-26 更健壮的专辑模式

```

// example-002/album.js
'use strict';
var mongoose = require('mongoose');

var trackSchema = mongoose.Schema({
  title: {type: String, required: true, index: true},
  duration: {
    required: true,
    type: {
      m: {type: Number, default: 0},
      s: {type: Number, default: 0}
    }
  }
});

var albumSchema = mongoose.Schema({
  composer: {type: String, required: true, index: true},
  title: {type: String, required: true, index: true},
  price: {type: Number, default: 0.0},
  releaseDate: {type: Date, default: Date.now},
  inPublication: Boolean,
  genre: {type: [String], index: true},
  tracks: [trackSchema]
});

albumSchema.index({composer: 1, title: 1});

albumSchema.path('tracks').validate(function (tracks) {

```

```

    return tracks.length > 0;
  }, 'Album has no tracks.');
```

```

var Album = mongoose.model('Album', albumSchema);

module.exports = Album;
```

11.3.7 模式引用

虽然 MongoDB 是非关系型数据库,但是文档集中文档之间的关系可以通过非正式的引用进行关联,这与外键的作用类似。当然,也可以把完整性和外键的问题完全留给应用解决。Mongoose 通过关联引用来构建模式之间的非正式关系,这种模式之间的链接使文档图能自动立即加载(也可以手动延迟加载)。由于用户可能建自己的个人专辑库,故要对这个音乐应用程序做拓展。由于专辑文档可以很大,每个专辑库文档中的数据最好避免重复。专辑库文档和独立的专辑文档之间可以创建引用,形成一种多对多的关系。当 Mongoose 加载专辑库文档时,一旦建立这些引用关系,返回的整个专辑库对象图中就包含了专辑文档。为简单起见,在 `example-003/library.json` 中定义了简单的专辑库。如清单 11-27 所示的专辑库,通过 `composer` 和 `title` 与专辑进行关联。当导入数据时,每个专辑必须通过文档 ID 与 MongoDB 中对应的专辑文档进行关联。

清单 11-27 专辑库的 JSON 数据

```

// example-003/library.json
{
  "owner": "Nicholas Cloud",
  "albums": [
    {
      "composer": "Kerry Muzzey",
      "title": "Renaissance"},
    {
      "composer": "Audiomachine",
      "title": "Tree of Life"},
    {
      "composer": "Jessica Curry",
      "title": "Dear Esther"}
  ]
}
```

如清单 11-28 所示,导入专辑库的脚本与导入专辑的脚本类似,但是还执行了一个重要的步骤。读入 `library.json` 文件并转化为 JavaScript 对象之后,将其中的专辑数据转化为 `example-001/import-albums.js` 中导入的实际的专辑文档对象。

清单 11-28 将专辑库数据导入 MongoDB

```

// example-003/import-library.js
'use strict';
var mongoose = require('mongoose');
var Album = require('./album-model');
var Library = require('./library-model');
var file2json = require('./file2json');
var fs = require('fs');
var path = require('path');

function handleError(err) {
  console.error(err);
}
```

```

    process.exit(1);
  }

  function resolveAlbums(libraryJSON, cb) {
    /*
     * [3] 使用联合$or 条件查询，查询多个 album 文件
     */
    var albumCriteria = {
      $or: libraryJSON.albums
    };

    Album.find(albumCriteria, cb);
  }

  mongoose.connect('mongodb://localhost/music');
  var db = mongoose.connection;
  db.on('error', handleError);
  db.once('open', function importLibrary () {

    /*
     * [1] 读取 library.json 文件数据并转换为普通 JavaScript 对象
     */
    var libraryFile = path.join(__dirname, 'library.json');
    file2json(libraryFile, 'utf8', function (err, libraryJSON) {
      if (err) return handleError(err);

      /*
       * [2] 查询与 library JSON 数据中的作曲家/标题相匹配的 album 文档
       */
      resolveAlbums(libraryJSON, function (err, albumDocuments) {
        if (err) return handleError(err);
        console.log('creating library');

        /*
         * [4] 将 album 文档赋值给 library 对象
         */
        libraryJSON.albums = albumDocuments;

        /*
         * [5] 根据 JSON 数据创建 library 文档并保存
         */
        var libraryDocument = new Library(libraryJSON);

        libraryDocument.save(function (err) {
          if (err) return handleError(err);
          process.exit(0);
        });
      });
    });
  });
}

```

清单 11-28 中对导入流的每个步骤做了注解，其中有几个步骤涉及前文中尚未介绍的概念。

在步骤[3]中，通过创建 `composer` 和 `title` 的 `or` 条件属性，现在只需了解 MongoDB 会检查所有的专辑文档并查看文档的 `composer/title` 与清单 11-29 中的 `or` 标准。

清单 11-29 专辑库引入 `$or` 标准

```

{ $or:
  [ { composer: 'Kerry Muzzey', title: 'Renaissance' },
    { composer: 'Audiomachine', title: 'Tree of Life' },
    { composer: 'Jessica Curry', title: 'Dear Esther' } ] }

```


在步骤[4]中, 查找到的专辑文档被赋值给 `libraryJSON.albums`, 覆盖了之前的 `composer/title` 数据。当保存专辑库文档时, Mongoose 将执行清单 11-30 中的专辑库模式。与之前的属性描述不同的是, 专辑属性的类型是含有 `ObjectIds` 数组的关联类型。在查询或保存专辑库文档时, `ref` 属性告诉 Mongoose 专辑文档可以被关联到该字段上。

清单 11-30 专辑库模式

```
// example-003/library-model.js
'use strict';
var mongoose = require('mongoose');

var librarySchema = mongoose.Schema({
  owner: String,
  albums: [{type: mongoose.Schema.Types.ObjectId, ref: 'Album'}]
});

var Library = mongoose.model('Library', librarySchema);
module.exports = Library;
```

Mongoose 文件会自动类型转换到相应的 `ObjectIds`。由于 Mongoose 能够自动进行类型转换, 所以把专辑文档添加到这个 `albums` 属性中时能通过模式的类型检查, 除这种方法外, 也可以通过导入脚本获取每个专辑文档的 `_id` 属性并放进 `albums` 数组里。两种方法的结果是完全相同的。

最后, 在步骤[5]中, 通过 `Library` 构造函数创建文档实例, 并将传入的原始 JSON 数据分配到每个文档属性上。也可以通过非构造函数的方法创建文档, 通过为实例的每个属性分配数据即可实现, 但是使用构造函数的方法是比较常用的。文档创建完成之后, `save()` 方法调用回调函数, 如果保存出错, 则会将错误信息传入回调函数。这与 `album` 导入脚本时使用模型静态函数 `create()` 创建多个专辑脚本是不同的。清单 11-31 显示了这种区别。

清单 11-31 创建单个文档与创建多个文档

```
// 一次创建一个文档
var libraryDocument = new Library(plainJSONLibrary);
libraryDocument.save(function (err) {...});

// 一次创建多个文档
Albums.create(arrayOfJSONAlbums, function (err) {...});
```

清单 11-32 中, 运行专辑库导入脚本与运行专辑导入脚本完全相同。

清单 11-32 运行专辑库导入脚本

```
example-003$ node import-library.js
creating library
```

一旦导入完成, 就可以通过 `mongo` 终端验证专辑库数据。清单 11-33 的输出显示, 通过把专辑转换为对应标识符, Mongoose 通过关联专辑标识符确实符合专辑库模式要求。(下一小节中介绍的使用模型与文档, 将研究如何使用模式引用属性快速加载所引用的文档。)

清单 11-33 在 MongoDB 中验证专辑库的导入

```
example-003$ mongo
MongoDB shell version: 2.6.7
connecting to: test
```

```
> use music
switched to db music
```

```
> db.libraries.find()
```

```
{ "_id" : ObjectId("54ed1dfdb11e8ae7252af342"), "owner" : "Nicholas Cloud", "albums" :
[ ObjectId("54ed1dcb6fb525ba25529bd1"), ObjectId("54ed1dcb6fb525ba25529bd7"),
ObjectId("54ed1dcb6fb525ba25529bf2") ], "_v" : 0 }
```

11.3.8 模式中间件

当验证、保存或删除特定的 MongoDB 文档时，Mongoose 会在模式对象上抛出事件，并且在每个操作之前和操作之后都抛出事件，可分别通过 `pre()` 和 `post()` 方法捕获前后两种事件。捕获事件是简单的函数，或者可以说是接收对应事件作为参数的中间件。`post` 事件中间件仅在事件完成之后监听文档，而 `pre` 事件中间件则在事件完成处理之前可能中断文档的生命周期。

在清单 11-34 中，对专辑库模式添加 `duration` 对象，与每个专辑曲目中的 `duration` 属性相同。不过这个 `duration` 对象将计算整个专辑库的总时长。为专辑库模式保存事件添加预处理中间件函数。在专辑库保存之前，该函数将遍历每个专辑和每个曲目，并对每首曲目的时长求和，将计算结果返回给 `duration` 对象。中间件函数仅接收一个参数，即回调函数 `next()`。当时长统计结束时，调用 `next()` 方法来完成关联在该模式上其他中间件的操作。

清单 11-34 预存储中间件

```
// example-004/library-model.js
'use strict';
var mongoose = require('mongoose');

var librarySchema = mongoose.Schema({
  owner: String,
  albums: [{type: mongoose.Schema.Types.ObjectId, ref: 'Album'}],
  duration: {
    h: {type: Number, default: 0},
    m: {type: Number, default: 0}
  }
});

librarySchema.pre('save', function (next) {
  var hours = 0, mins = 0;
  /*
   * 遍历所有 album 并添加 hours 和 minutes 字段
   */
  this.albums.forEach(function (album) {
    album.tracks.forEach(function (track) {
      hours += track.duration.h;
      mins += track.duration.m;
    });
  });
  /*
   * 将分钟数除以 60 得到小时数，将剩余的分钟数重新赋值给 mins 字段
   */
  hours += (mins / 60);
  mins = (mins % 60);
  this.duration = {h: hours, m: mins};
  next();
});

var Library = mongoose.model('Library', librarySchema);

module.exports = Library;
```

可以同步或异步执行预处理事件中间件。清单 11-34 中的代码是同步的，也就是说，其他代码必

须在时长统计结束之后才可以执行。另外，可以向 `pre()` 方法传递一个额外的布尔类型参数，标记该处理函数是异步的，改变原来一个接一个的执行方法。

中间件函数本身也支持 `done()` 参数，如清单 11-35 所示。在同步方法里，当前中间件函数执行完成并调用 `next()` 方法时，才能调用后一个中间件函数。在异步方法里也同样如此，只是 `done()` 函数肯定会在未来的下一个事件循环中异步操作完成时才能调用。清单 11-35 的执行顺序如下。

1. 执行时长求和处理，为下一个事件循环做准备。
2. 调用 `next()` 方法，调用下一个中间件。
3. 在未来的某个时间点，通过调用 `done()` 来标识该中间件的操作已完成。

清单 11-35 异步调用预处理中间件

```
// example-005/library-model.js
// ...
librarySchema.pre('save', true, function (next, done) {

  var hours = 0, mins = 0;
  process.nextTick(function () {
    /*
     * 遍历所有 album 并添加 hours 和 minutes 字段
     */
    this.albums.forEach(function (album) {
      album.tracks.forEach(function (track) {
        hours += track.duration.h;
        mins += track.duration.m;
      });
    });
    /*
     * 将分钟数除以 60 得到小时数，将剩余的分钟数重新赋值给 mins 字段
     */
    hours += (mins / 60);
    mins = (mins % 60);
    this.duration = {h: hours, m: mins};
    done();
  });

  next();
});

var Library = mongoose.model('Library', librarySchema);
module.exports = Library;
```

如果预事件处理中间件函数在同步执行中报错，错误信息将作为唯一参数传递给 `next()` 函数。然而，如果在异步方法中报错，错误信息将传递给 `done()` 函数。给回调函数传递错误信息将导致该事件执行失败，并将错误信息传递到最终操作的回调函数中，如传递到文档的 `save()` 方法。

post 事件中间件函数不接收控制流参数，但是在事件处理完成后接收一份文档的副本。

11.4 使用模型和文档

Mongoose 模型是创建文档实例的构造函数。它所创建的实例遵从 Mongoose 模式，并提供一系列维护文档的方法集。模型与 MongoDB 集合相关联。事实上，当保存 Mongoose 文档时，如果其对应的方法集不存在，Mongoose 将创建这些方法集。按照惯例，以名词的单数形式命名模型（如 Album），

以复数形式命名集合（如 albums）。

通过调用 `mongoose.model()`，传入模型名称与模型模式作为参数即可创建模型构造函数。

所有通过此方法创建的文档，无论是通过代码还是 `Mongoose` 执行查询时，返回的文档实例都遵从 `Mongoose` 模式。清单 11-36 显示了在 `MongoDB` 中创建 `album` 文档的导入脚本中用于创建 `Album` 构造函数的代码。

清单 11-36 专辑模型

```
// example-006/album-model.js

//...模式定义...

var Album = mongoose.model('Album', albumSchema);

module.exports = Album;
```

在通过 `mongoose.model()` 方法注册 `Mongoose` 模型后，如果相关的模式属性有引用关系，`Mongoose` 就可以通过名称解析该模型。如清单 11-37 所示，前面的例子使用模型方法创建专辑库模式和 `Album` 模型之间的引用关系。

清单 11-37 专辑库模式引用专辑模型

```
// example-006/library-model.js

// ...
var librarySchema = mongoose.Schema({
  // ...
  albums: [{type: mongoose.Schema.Types.ObjectId, ref: 'Album'}],
  // ...
});
```

通过模型构造函数可以创建文档，通过模型查询方法从 `MongoDB` 数据存储中也能获取文档。每个文档可以从 `MongoDB` 集合自行保存或删除。这一点与关系型数据库（`RMDB`）中常见的活动记录（`ActiveRecord`）数据访问模式非常类似。清单 11-38 中，通过 `Album` 构造函数创建了新的 `album` 文档实例。`Album` 数据赋值给由专辑模式定义的每个属性。最后，调用 `save()` 方法，并在 `MongoDB` 中创建相关文档后调用其回调函数。

清单 11-38 创建并保存新的文档实例

```
// example-006/add-album-instance.js
'use strict';
var mongoose = require('mongoose');
var Album = require('./album-model');
function handleError(err) {
  console.error(err);
  process.exit(1);
}

mongoose.connect('mongodb://localhost/music');
var db = mongoose.connection;
db.on('error', handleError);
db.once('open', function addAlbumInstance() {

  var album = new Album();
  album.composer = 'nervous_testpilot';
  album.title = 'Frozen Synapse';
  album.price = 8.99;
  album.releaseDate = new Date(2012, 8, 6);
  album.inPublication = true;
  album.genre = ['Dance', 'DJ/Electronica', 'Soundtrack'];
```



```

album.tracks = [
  {
    title: 'Welcome to Markov Geist',
    duration: {m: 1, s: 14}},
  // ...additional tracks...
];

album.save(function (err) {
  if (err) return handleError(err);
  console.log('album saved', album);
  process.exit(0);
});
});

```

保存专辑后，脚本输出文档数据如下。

```

example-006$ node add-album-instance.js
album saved { _v: 0,
  inPublication: true,
  title: 'Frozen Synapse',
  composer: 'nervous testpilot',
  _id: 54f117e4a27cc5375e156c6d... }

```

事实上，如清单 11-39 所示，可以通过 MongoDB 查询验证所创建的专辑集。

清单 11-39 验证在 MongoDB 中创建的 Mongoose 文档

```

example-006$ mongo
MongoDB shell version: 2.6.7
connecting to: test
> use music
switched to db music
> db.albums.find({composer: 'nervous_testpilot'}, {_id: 1, composer: 1, title: 1})
{ "_id" : ObjectId("54f117e4a27cc5375e156c6d"), "title" : "Frozen Synapse",
  "composer" : "nervous_testpilot" }

```

也可以通过直接向模型构造函数传入对象来设置文档实例的属性。当文档数据在 JavaScript 对象中已经存在时，这种方法非常有用。例如，从 Web 请求主题中反序列化得到的 JSON 数据或从文件解析出的 JSON 数据。清单 11-40 采用了前文中的案例从 JSON 文件加载得到专辑数据，然后使用专辑模型构造函数通过 JSON 数据创建文档。由于 JSON 数据符合专辑模式（或者就像示例中一样，`releaseDate` 的日期字符串类型可直接转换为模式属性类型 `Date`），所以专辑实例能够正确的被保存。

清单 11-40 使用属性数据创建一个文档的可用方法

```

// example-007/add-album-instance-alt.js
'use strict';
var mongoose = require('mongoose');
var Album = require('./album-model');
var file2json = require('./file2json');
var path = require('path');

function handleError(err) {
  console.error(err);
  process.exit(1);
}

mongoose.connect('mongodb://localhost/music');
var db = mongoose.connection;
db.on('error', handleError);
db.once('open', function addAlbumInstance() {

  var albumFile = path.join(__dirname, 'album.json');
  file2json(albumFile, 'utf8', function (err, albumJSON) {
    var album = new Album(albumJSON);

```

```
album.save(function (err) {
  if (err) return handleError(err);
  console.log('album saved', album);
  process.exit(0);
});
});
});
```

11.4.1 文档实例方法

文档不仅仅包含数据，还可以包含用户行为。当创建文档实例时，Mongoose 创建原型链，该原型链具有模式对象中定义在 `method` 属性上的方法副本。以这种方式定义的文档方法能通过 `this` 关键字存取特殊的文件实例。

清单 11-41 显示了专辑模式定义的两个实例方法：一个方法是根据前一首曲目标题查找下一首曲目，另一个方法是根据共同的音乐流派查找类似的专辑。`findSimilar()` 方法中使用的查询语法将在使用查询语句一节中详细介绍，现在仅需了解该语法用于高效地查找与专辑实例具有相同音乐流派的专辑，并且不与专辑实例具有相同的 `_id`（所以查询结果列表中不包含其本身）。

清单 11-41 在模式中定义文档实例方法

```
// example-008/album-model.js

// ...
var albumSchema = mongoose.Schema({/*...*/});

albumSchema.methods.nextTrack = function (previousTrackTitle) {
  var i = 0, len = this.tracks.length;
  for (i; i < len; i += 1) {
    if (this.tracks[i].title !== previousTrackTitle) {
      continue;
    }
    // return the next track, or, if this is the last track,
    // return the first track
    return this.tracks[i + 1] || this.tracks[0];
  }
  throw new Error('unable to find track ' + previousTrackTitle);
};

albumSchema.methods.findSimilar = function (cb) {
  var criteria = {
    _id: {$ne: this._id},
    genre: {$in: this.genre}
  };
  this.model('Album').find(criteria)
    .exec(cb);
};

var Album = mongoose.model('Album', albumSchema);

module.exports = Album;
```

清单 11-42 中的脚本加载标题为 *Renaissance* 的专辑，然后调用 `album.nextTrack()` 方法查找曲目 *Fall from Grace* 的下一首曲目。再调用 `album.findSimilar()` 方法加载 *Renaissance* 引用的专辑，将这些专辑的标题与类型在终端打印出来。从输出结果可以看出，这些专辑的类型均相同，专辑 *Renaissance* 本身并不包含在查找结果之内。

清单 11-42 使用文档实例方法
// example-008/index01.js

```

'use strict';
var mongoose = require('mongoose');
var Album = require('./album-model');

function handleError(err) {
  console.error(err);
  process.exit(1);
}

mongoose.connect('mongodb://localhost/music');
var db = mongoose.connection;
db.on('error', handleError);
db.once('open', function () {
  Album.findOne({title: 'Renaissance'})
    .exec(function (err, album) {
      if (err) return handleError(err);

      var nextTrack = album.nextTrack('Fall from Grace');
      console.log('next track:', nextTrack.title);

      album.findSimilar(function (err, albums) {
        if (err) return handleError(err);
        console.log('this album:', album.title, album.genre);
        albums.forEach(function (album) {
          console.log('similar album:', album.title, album.genre);
        });
        process.exit(0);
      });
    });
});

```

```
example-008$ node index01.js
```

```

next track: Fall from Grace (Choir Version)
this album: Renaissance ["Classical","Trailer Music","Soundtrack"]
similar album: Tree of Life ["Classical","Trailer Music"]
similar album: Dear Esther ["Classical","Video Game Soundtrack"]
similar album: Frozen Synapse ["Dance","Electronica","Soundtrack"]

```

11.4.2 文档虚拟属性

与实例方法类似的是，虚拟 `getter` 和 `setter` 属性可通过模式添加到文档中。这些虚拟属性的行为与普通的数据属性类似，但是当文档保存后它们并不是持久化的。这些方法对于基于文档数据的计算和返回、数据解析、数据转换等有很大的帮助。清单 11-43 中使用 `getter` 和 `setter` 方法为专辑模式添加虚拟方法 `composerInverse`，`composerInverse` 方法以姓+名的顺序获取作曲家的名字，并设置为倒序后的名+姓这样的顺序。

清单 11-43 中使用虚拟 `getter` 和 `setter` 向专辑模式中添加虚拟属性 `composerInverse`，从 `composerInverse` 取值会得到倒序以姓+名的顺序的作曲家名字，向 `composerInverse` 赋值一个倒序形式的名字则会得到正确的名+姓这样的顺序。

清单 11-43 文档虚拟属性

```
// example-08/album-model.js
```

```

var albumSchema = mongoose.Schema({/*...*/});

// ...
albumSchema.virtual('composerInverse').get(function () {
  var parts = this.composer.split(' '); //first last

```

```

    if (parts.length === 1) {
      return this.composer;
    }
    return [parts[1], parts[0]].join(', '); //last, first
  });
}

albumSchema.virtual('composerInverse').set(function (inverse) {
  var parts = inverse.split(', '); //last, first
  if (parts.length === 1) {
    this.composer = inverse;
  }
  this.composer = [parts[1], parts[0]].join(' '); //first last
});
// ...

```

传递到 `Schema.virtual()` 方法的字符串参数定义了文档实例创建后, 该属性所在的文档路径。通过指定从文档根路径开始的完整路径, 也可以将文档虚拟属性赋值给子文档和嵌套对象。例如, 假设作曲家属性是含有 `firstName` 与 `lastName` 属性的对象, 虚拟属性将作用在 `composer.inverse` 方法上。

清单 11-44 的代码和输出结果显示了 `composerInverse` 属性的行为。

清单 11-44 获取和设置虚拟属性

```

// example-008/index02.js
'use strict';
var mongoose = require('mongoose');
var Album = require('./album-model');

function handleError(err) {
  console.error(err);
  process.exit(1);
}

mongoose.connect('mongodb://localhost/music');
var db = mongoose.connection;
db.on('error', handleError);
db.once('open', function () {
  Album.find({}).exec(function (err, albums) {
    if (err) return handleError(err);

    albums.forEach(function (album) {
      console.log('album.composer:', album.composer);
      var inverse = album.composerInverse;
      console.log('album.composerInverse:', inverse);
      album.composerInverse = inverse;
      console.log('album.composer:', album.composer);
      console.log(/*newline*/);
    });

    process.exit(0);
  });
});

example-008$ node index02.js
album.composer: Kerry Muzze
album.composerInverse: Muzze, Kerry
album.composer: Kerry Muzze

album.composer: Audiomachine
album.composerInverse: Audiomachine
album.composer: Audiomachine

album.composer: Jessica Curry
album.composerInverse: Curry, Jessica
album.composer: Jessica Curry

album.composer: nervous_testpilot

```



```
album.composerInverse: nervous_testpilot
album.composer: nervous_testpilot
```

11.4.3 静态模型方法

可以为模型（而不是文档实例）添加静态方法，这在集合中查询中封装复杂的标准构造函数时是常用的方法。清单 11-45 中，为专辑模型的静态属性添加 `inPriceRange()` 方法。`inPriceRange()` 接收价格下限和上限作为参数，并查找在该价格区间之内的专辑。

清单 11-45 为模型添加静态方法

```
// example-009/album-model.js
```

```
var albumSchema = mongoose.Schema({/*...*/});
```

```
// ...
```

```
albumSchema.statics.inPriceRange = function (lower, upper, cb) {
```

```
  var criteria = {
```

```
    price: {$gte: lower, $lte: upper}
```

```
  };
```

```
  this.find(criteria)
```

```
    .exec(cb);
```

```
};
```

```
// ...
```

通过模式创建专辑模型之后，任何静态方法都将与模型绑定。虽然在实例方法中，`this` 值指向文档本身，但是在静态方法中，`this` 值指向模型构造函数（如 `Album`）。任何可以在模型上调用的方法如 `find()` 和 `create()`，均可以在静态方法中访问。

清单 11-46 中，在命令行中传入两个价格作为运行参数，查找该价格区间之内的专辑。与其他静态方法类似，`inPriceRange()` 方法是在专辑模型上调用的。由于模型的查询逻辑相对独立，为了避免影响应用程序的其他部分，以这种方式封装查询来处理独立性问题是一种很好的方法。

清单 11-46 使用静态模型方法

```
'use strict';
```

```
var mongoose = require('mongoose');
```

```
var Album = require('./album-model');
```

```
var lower = Number(process.argv[2] || 0);
```

```
var upper = Number(process.argv[3] || lower + 1);
```

```
console.log('finding albums between $%s and $%s', lower.toFixed(2), upper.toFixed(2));
```

```
function handleError(err) {
```

```
  console.error(err);
```

```
  process.exit(1);
```

```
}
```

```
mongoose.connect('mongodb://localhost/music');
```

```
var db = mongoose.connection;
```

```
db.on('error', handleError);
```

```
db.once('open', function () {
```

```
  Album.inPriceRange(lower, upper, function (err, albums) {
```

```
    if (err) return handleError(err);
```

```
    console.log('found albums:', albums.length);
```

```
    albums.forEach(function (album) {
```

```
      console.log(album.title, '$' + album.price.toFixed(2));
```

```
    });
```

```

    process.exit(0);
  });
});

example-009$ node index.js 5.00 10.00
finding albums between $5.00 and $10.00
found albums: 3
Tree of Life $9.49
Dear Esther $6.99
Frozen Synapse $8.99

example-009$ node index.js 9.00 10.00
finding albums between $9.00 and $10.00
found albums: 1
Tree of Life $9.49

example-009$ node index.js 20.00
finding albums between $20.00 and $21.00
found albums: 0

```

■ **提示** 为了简化示例，在下一小节的查询示例中并未使用封装静态模型方法。但是在实际的应用程序维护中，不用静态模型方法封装查询是很糟糕的实践。

11.5 使用查询

Mongoose 查询是由零个或多个属性指定查询参数的普通对象。空查询对象匹配所有结果。Mongoose 条件查询对象支持 MongoDB 查询语法。模型提供了几种使用条件对象的查询方法，用于过滤并返回 Mongoose 文档。在后续示例中，Web 服务器支持通过 Mongoose 模型访问 MongoDB 数据。

开启 Web 服务器之前，需确保 MongoDB 处于运行状态，然后在示例代码所在目录下，在命令行中执行清单 11-47 中的命令。（每个示例代码的第一行均显示了该示例代码所在的目录路径。）运行结果显示 Web 服务器运行在 8080 端口上。与服务器的交互将显示在可用于大多数平台的 curl 终端工具上，每个示例均可运行在标准 HTTP 客户端上。

清单 11-47 开启在示例 10 中的 Web 服务器

```

example-XYZ$ node index.js
listening on port 8080

```

11.5.1 Model.find()

基本的 CRUD（增删改查）操作与对应的 Mongoose 模型函数一一匹配。例如，通常使用 `Album.find()` 查找与条件（criteria）对象匹配的专辑，清单 11-48 中示例使用了路由，该通用路由使用 `Album.find()` 查找与条件对象匹配的专辑。如果 URL 请求中含有 `composer` 和 `title` 参数，条件对象从 URL 的查询字段中获取 `composer` 和 `title` 参数。如果条件对象中设置了两个参数或其中一个参数，Mongoose 将仅返回与这些属性相匹配的专辑文档（与 SQL 条件查询类型）。如果条件对象中并未设置参数，则条件对象为空，Mongoose 将返回所有专辑文档。

清单 11-48 查找与给定的条件匹配的专辑

```

// example-010/album-routes.js

/**

```

```

* GET /album(?composer={string}&title={string})
* @param req
* @param cb
*/
routes.GET['^\\album(?:\\?.+)?$',] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var criteria = {};
  if (req.query.composer) {
    criteria.composer = req.query.composer;
  }
  if (req.query.title) {
    criteria.title = req.query.title;
  }
  Album.find(criteria)
    .sort({composer: 1, title: 1})
    .lean(true)
    .exec(function (err, albums) {
      if (err) return cb(500, err);
      cb(200, albums);
    });
};

```

`Album.find()`方法将返回 Mongoose 查询对象，该对象提供对查找结果的其他操作方法。

提示 可以使用多种方法调用模型方法。第一种方法，如清单 11-48 所示，当 `Query.exec()` 方法调用完成之后，返回的查询对象提供一个流畅的接口支持以链式调用的方式查询选项。第二种方法避免将查询对象组合在一起。如果回调函数作为最后一个参数传递给模型查询方法（如 `find({, function () {...}})`），默认立即执行查询操作并将结果传递给回调函数。对于简单查询来说，第二种方法更为简洁。

第一个查询指令是 `Query.sort()`，支持使用 MongoDB 的排序符号。这些属性会通知 MongoDB 哪些属性需要排序、使用升序还是降序（1 为升序，-1 为降序）。清单 11-48 的代码执行结果将先按 `composer` 升序排序，再按 `title` 升序排序。

在 `Query.sort()` 方法之后，调用 `Query.lean()` 方法通知 Mongoose 以 JSON 数据返回结果，而不是返回 Mongoose 文档作为结果。

默认情况下，Mongoose 返回文档，该文档具有 Mongoose 特定属性和方法，可用于校验、保存以及其他处理文档对象。由于 `route` 只是简单将结果序列化并返回给客户端，将结果以 JavaScript 对象（或 JSON 对象）返回更为合适。

一旦查询结果准备好，`exec()` 方法的回调函数接收错误信息作为参数，或是接收 `Album.find()` 操作返回的数据作为参数。返回的结果是与查询条件（如果有）相匹配的专辑对象数组。清单 11-49 显示了几个使用多种字符串查询参数的 `curl` 命令。每个案例的输出结果都是 Web API 返回的序列化 JSON 数组。

提示 以下示例使用作者的电脑上的 MongoDB 标识符。读者电脑上的标识符或许不同。读者可以使用 `mongo` 终端查看标识符，查看方法如前文所述。

清单 11-49 使用 curl 多条件查找

```

example-010$ curl -X GET http://localhost:8080/album?composer=Kerry%20Muzzey
[{"_id":"54ed1dcb6fb525ba25529bd1","composer":"Kerry Muzzey","title":"Renaissance"... ]

```

```
example-010$ curl -X GET http://localhost:8080/album?title=Dear%20Esther
[{"_id":"54ed1dcb6fb525ba25529bf2","composer":"Jessica Curry","title":"Dear Esther"... ]
example-010$ curl -X GET
"http://localhost:8080/album?composer=Audiomachine&title=Tree%20of%20Life"
[{"_id":"54ed1dcb6fb525ba25529bd7","composer":"Audiomachine","title":"Tree of Life"... ]
```

1. Model.findById()

Album.find() 方法的返回结果一定是数组（即使与查询条件仅匹配一条记录），Album.findById() 方法仅返回一条与查询条件匹配的记录。清单 11-50 通过 albumID 参数获取专辑。albumID 参数不是从查询字符串获取，而是从 URL 上最后一个部分获取的。查询结果返回后再调用 lean() 方法，去除查询结果的 MongoDB 文档实例中非必要的属性和方法。

清单 11-50 查找与给定条件匹配的专辑

```
// example-010/album-routes.js
```

```
/**
 * GET /album/{id}
 * @param req
 * @param cb
 */
routes.GET['^/album/([a-z0-9]+)$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var albumID = req.params[0];
  Album.findById(albumID)
    .lean(true)
    .exec(function (err, album) {
      if (err) return cb(500, err);
      cb(200, album);
    });
};
```

```
example-010$ curl -X GET http://localhost:8080/album/54f3a4df056601726f867685
{"_id":"54f3a4df056601726f867685","composer":"nervous_testpilot","title":"Frozen
Synapse"... }
```

前文中通过导入脚本 example-007/add-album-instance-alt.js 创建新的专辑，传递非序列化 JSON 对象至 Album 构造函数创建 album 实例。清单 11-51 再次展示了该处理方法。将序列化专辑数据转换为 JSON 对象，然后传递给 Album 模型构造函数。一旦文档实例创建完成，save() 方法就使用 album 模式中定义的规则对数据进行校验，并创建新的 MongoDB 文档。

清单 11-51 创建新的专辑文档

```
// example-010/album-routes.js
```

```
/**
 * POST /album
 * @param req
 * @param cb
 */
routes.POST['^/album$'] = function (req, cb) {
  console.log(req.body);
  cb = httpd.asJSON(cb);
  var albumJSON = req.body;
  var album = new Album(albumJSON);
```



```
album.save(function (err) {
  if (err) return cb(500, err);
  cb(201, album.toObject());
});
};
```

如果校验失败，或是专辑无法创建完成，则会将错误信息传递给回调函数，再作为 HTTP 500 内部服务器错误发送到客户端。如果专辑文档创建成功，会将序列化的 JSON 数据发送到客户端。与之前的 routes 不同的是，之前的 `Query.lean()` 用于确保只有数据被序列化，此处只有调用 `toObject()` 方法，才能以 JSON 格式返回文档自身的数据，这是与在查询链中的 `lean()` 等价的手动方法。

清单 11-52 中的 curl 请求读取文件 `example-010/new-album.json` 中的内容，并设置为请求体。请求头中的 `Content-Type` 信息通知 Web 服务器如何解析请求的内容。

清单 11-52 使用 curl 请求创建专辑

```
example-010$ curl -X POST http://localhost:8080/album\
> -d @new-album.json\
> -H "Content-Type: application/json"
{"_id":"54f66ed2fa4af12b43fee49b","composer":"Aphelion","title":"Memento"... }
```

`example-010/new-album.json` 中的专辑数据缺少 `releaseDate` 属性，但并未导致模式校验失败，这是因为 `releaseDate` 是非必要属性。实际上，`releaseDate` 属性的默认值是 `Date.now`，通过 mongo 终端查询可以看到 `releaseDate` 属性确实使用了默认值。但是，专辑并不都是今天发行的，所以有必要更新该专辑文档中对应的数据。

2. Model.findByIdAndUpdate()

可以通过诸多方法更新专辑实例。`Album.findById()` 方法可以获取文档，将其对应属性设置为更新数据，并可被保存到数据库。`Album.findByIdAndUpdate()` 方法则可直接完成查找及更新的操作，并返回更新之后的专辑文档，如清单 11-53 所示。

清单 11-53 通过 ID 查找与更新专辑

```
// example-010/album-routes.js
/**
 * PUT /album/{id}
 * @param req
 * @param cb
 */
routes.PUT['^/album/([a-z0-9]+)$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var albumID = req.params[0];
  var updatedFields = req.body;
  Album.findByIdAndUpdate(albumID, updatedFields)
    .lean(true)
    .exec(function (err, album) {
      if (err) return cb(500, err);
      cb(200, album);
    });
};
```

如清单 11-51 所示，通过 HTTP 请求发送序列化 JSON 对象。该请求是 PUT 请求，在 URL 中包含 album 标识符。该请求体中发送的数据是用于更新的数据。无须发送整个文档，Mongoose 会进行增量更新。一旦请求体解序列化后，专辑 ID 和更新过的字段将传递给 `findByIdAndUpdate()`。如

果更新成功，并且无错误产生，更新之后的文档将传递给最终的查询回调函数。

清单 11-54 显示了 curl 命令，使用简单的 JSON 创建 PUT 请求用于更新 releaseDate 字段。当请求结束，将返回的更新结果打印出来。

清单 11-54 使用 curl 通过 ID 查询并更新专辑

```
example-010$ curl -X PUT http://localhost:8080/album/54f66ed2fa4af12b43fee49b \
> -d '{"releaseDate": "2013-08-15T05:00:00.000Z"}' \
> -H "Content-Type: application/json"
{"_id": "54f66ed2fa4af12b43fee49b" ... "releaseDate": "2013-08-15T05:00:00.000Z" ... }
```

3. Model.findByIdAndRemove()

为了从 MongoDB 中删除文档，可以通过路由的 DELETE，使用 Album.findByIdAndRemove() 查询 MongoDB 文档并将其从专辑集中删除。如清单 11-55 所示，如果操作成功，被删除的专辑将被传递给最终的回调函数。

清单 11-55 通过 ID 查找专辑并删除

```
//example-010/album-routes.js
/**
 * DELETE /album/{id}
 * @param req
 * @param cb
 */
routes.DELETE['^/album/([a-z0-9]+)$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var albumID = req.params[0];
  Album.findByIdAndRemove(albumID)
    .lean(true)
    .exec(function (err, album) {
      if (err) return cb(500, err);
      cb(200, album);
    });
};

example-010$ curl -X DELETE http://localhost:8080/album/54f3aa9447429f44763f2603
{"_id": "54f66ed2fa4af12b43fee49b", "composer": "Aphelion", "title": "Memento" ... }
```

文档实例也有 remove() 方法，可在 save() 等方法中调用。清单 11-56 中，通过 ID 获取一个专辑实例。此处未使用 Query.lean() 方法，因为这是文档而非 JSON 数据，该文档具有 remove() 方法。获取实例之后，通过回调函数调用 remove()。如果删除失败，返回错误信息；如果删除成功，则返回所删除文档的副本。

清单 11-56 删除文档实例

```
Album.findById(albumID)
  .exec(function (err, albumInstance) {
    albumInstance.remove(function (err, removedAlbum) {
      // album has been removed
    });
  });
```

4. Model.count()

另一种有用的模型方法是 count()，与 find*() 方法接收相同的条件对象类型，但返回结果是对象的数目而不是整个对象。清单 11-57 中，使用和普通查询专辑相同的查询参数搜索，返回满足条件的记录数。

清单 11-57 统计满足条件的专辑数目

// example-011/album-routes.js

```
/**
 * GET /album/count(?composer={string}&title={string})
 * @param req
 * @param cb
 */
routes.GET['^\\album\\/count(?:\\?.+)?$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var criteria = {};
  if (req.query.composer) {
    criteria.composer = req.query.composer;
  }
  if (req.query.title) {
    criteria.title = req.query.title;
  }
  Album.count(criteria)
    .exec(function (err, count) {
      if (err) return cb(500, err);
      cb(200, count);
    });
};
```

example-011\$ curl -X GET http://localhost:8080/album/count

4

example-011\$ curl -X GET http://localhost:8080/album/count?composer=Jessica%20Curry

1

5. Query.Populate()

前文在清单 11-28 中使用脚本添加音乐专辑库到 MongoDB 中。清单 11-58 中，专辑库模式定义了一个数组属性 `albums`，包含对专辑文档的引用。

清单 11-58 在专辑库模式中关联专辑

```
var librarySchema = mongoose.Schema({
  // ...
  albums: [{type: mongoose.Schema.Types.ObjectId, ref: 'Album'}],
  // ...
});
```

不论是否解析其他文档对象的引用，具有外部引用的 Mongoose 文档都可以获取到。清单 11-59 中的路由通过 ID 获取专辑库，然后调用 `Query.populate()` 方法即时获取相关的专辑。Mongoose 知道尽管 `albums` 是数组，但它包含的对象实际上是引用了其他专辑文档。

清单 11-59 使用库模型填充专辑

// example-011/library-routes.js

```
/**
 * GET /library/(id)
 * @param req
 * @param cb
 */
routes.GET['^\\library\\/([a-z0-9]+)$'] = function (req, cb) {
```

```

cb = httpd.asJSON(cb);
var libraryID = req.params[0];
Library.findById(libraryID)
  .populate('albums')
  .lean(true)
  .exec(function(err, library) {
    if (err) return cb(500, err);
    if (!library) return cb(404, {
      message: 'no library found for ID ' + libraryID
    });
    cb(200, library);
  });
}

```

图 11-1 显示了格式化的 HTTP 响应。在 albums 中的每个专辑已被完全解引用。由于在查询链中调用了 Query.lean(), Mongoose 将 library 和 album 的数据转化为 JSON 对象。

```

{
  "_id": "54ed249312c06b3726d3abed",
  "owner": "Nicholas Cloud",
  "albums": [
    {
      "_id": "54ed1dcb6fb525ba25529bd1",
      "composer": "Kerry Muzzey",
      "title": "Renaissance",
      "price": 4.95,
      "releaseDate": "2014-01-13T06:00:00.000Z",
      "inPublication": true,
      "tracks": [
        {
          "title": "The Looking Glass",
          "_id": "54ed1dcb6fb525ba25529bd6",
          "duration": {
            "m": 3,
            "s": 10
          }
        }
      ],
      "genre": [
        "Classical",
        "Trailer Music",
        "Soundtrack"
      ],
      "_v": 0
    },
    {
      "_id": "54ed1dcb6fb525ba25529bd7",
      "composer": "Audiomachine",
      "title": "Tree of Life",
      "_v": 0
    },
    {
      "_id": "54ed1dcb6fb525ba25529bf2",
      "composer": "Jessica Curry",
      "title": "Dear Esther",
      "_v": 0
    }
  ]
}

```

图 11-1 专辑库填充结果

11.5.2 使用查询运算符查找文档

到目前为止，album 与 library 路由由基本的增删改查操作（CRUD）组成，形成基本的 Web API，但是还可以使 API 更健壮。MongoDB 支持一系列有用的查询运算符，可用于以特定的方式过滤数据。

1. \$lt 与 \$gt 运算符

\$lt 与 \$gt 运算符可用于查找某些属性值小于 (\$lt) 或大于 (\$gt) 指定值的文档。清单 11-60 中，允许客户端搜索发行日期在由查询参数设定的日期之前或之后的专辑。

```

清单 11-60 通过发行日期查找专辑
// example-011/album-routes.js

/**
 * GET /album/released/MM-DD-YYYY

```



```

* GET /album/released/MM-DD-YYYY/before
* GET /album/released/MM-DD-YYYY/after
* @param req
* @param cb
*/
routes.GET(/^\/album\/released\/([\d]{2}-[\d]{2}-[\d]{4})(?:\/(before|after))?$' ] =
function (req, cb) {
  cb = httpd.asJSON(cb);
  var date = req.params[0];
  var when = req.params[1];

  var criteria = {releaseDate: {}};
  if (when === 'before') {
    criteria.releaseDate.$lt = new Date(date);
  } else if (when === 'after') {
    criteria.releaseDate.$gt = new Date(date);
  } else {
    when = null;
    criteria.releaseDate = new Date(date);
  }

  Album.find(criteria)
    .select('composer title releaseDate')
    .lean(true)
    .exec(function (err, albums) {
      if (err) return cb(500, err);
      if (albums.length === 0) {
        return cb(404, {
          message: 'no albums ' + (when || 'on') + ' release date ' + date
        });
      }
      cb(200, albums);
    });
};

```

为了查找在某个指定日期发行的专辑，使用条件对象映射日期值与 `releaseDate` 属性：

```
{releaseDate: new Date(...)}
```

如果搜索在某个日期之前或之后发行的专辑，则分别使用 `$lt` 或 `$gt` 运算符：

```
{releaseDate: {$lt: new Date(...)}}
```

```
// or
```

```
{releaseDate: {$gt: new Date(...)}}
```

查找发行日期在某个日期之前或等于某个日期的专辑，可以使用 `$lte` 运算符。类似地，`$gte` 运算符可用于查找发行日期从某个指定日期开始的专辑。查找发行日期不在某一天的专辑可以使用 `$ne` 运算符。`$ne` 的逆运算是 `$eq`，如果单独使用 `$eq`。其作用与直接设置条件对象中 `releaseDate` 的值等价。为保证响应尽可能小，在查询开始之前调用 `Query.select()` 方法。该方法限制了返回结果的属性。在本示例中，查询语句仅选择 `composer`、`title` 和 `releaseDate`，均包含在空格分隔的字符串内，忽略其他属性。清单 11-61 显示了按发行日期查询返回的 JSON 数据。

清单 11-61 使用 curl 通过发行日期查找专辑

```
example-011$ curl -X GET http://localhost:8080/album/released/01-01-2013
{"message":"no albums on release date 01-01-2013"}
```

```
example-011$ curl -X GET http://localhost:8080/album/released/01-01-2013/before
[{"_id":"54ed1dc6fb525ba25529bf2","composer":"Jessica Curry","title":
"DearEsther","releaseDate":"2012-02-14T06:00:00.000Z"}, {"_id":"54f3a4df056601726f867685",
"composer":"nervous_testpilot","title":"Frozen Synapse","releaseDate":"2012-09-
06T05:00:00.000Z"}]
```

```
example-011$ curl -X GET http://localhost:8080/album/released/01-01-2013/after
[{"_id":"54ed1dc6fb525ba25529bd1","composer":"Kerry Muzzey","title":"Renaissance",
"releaseDate":"2014-01-13T06:00:00.000Z"}, {"_id":"54ed1dc6fb525ba25529bd7","composer":
"Audiomachine","title":"Tree of Life","releaseDate":"2013-07-16T05:00:00.000Z"}]
```

注意，尽管其中的 `Query.select()` 并没有指定排除 `_id` 属性，它依然展示在了每次响应中，若想删除这个属性，需要在选择字符串前面加上一个负号（-）。为 `_id` 属性加上负号前缀能防止这个属性被选择出来：

```
Album.find(...)
  .select('-_id composer title releaseDate')
// ...
```

提示 当按指定的属性获取包含关系时，`_id` 属性是唯一可用于排除的属性，否则，包含和排除属性无法混合使用。查询语句仅可以选择特定的属性，或仅排除特定的属性，或同时选择与排除某些属性。如果 `Query.select()` 字符串中有任何属性是否定的（`_id` 除外），所有指定的属性必须否定，否则将抛出错误。

2. \$in 与 \$nin 运算符

通过某子集匹配的属性值，对于筛选文档是非常有用的。`$in` 运算符（逆运算 `$nin`）将文档属性与数组中的每个元素进行测试。如果文档属性与数组中至少有一个元素相匹配，那么文档就满足筛选条件。例如，查询两位作曲家的专辑可使用清单 11-62 中的条件对象。

清单 11-62 使用 `$in` 查询运算符筛选作曲家

```
{composer: {$in: ['Kerry Muzzey', 'Jessica Curry']}}
```

`$nin` 运算符与 `$in` 运算符恰好相反，如果属性值不在集合中，则筛选符合条件。`$in` 与 `$nin` 运算符均可作用于标量属性（如字符串、数值、日期等），但是这两个运算符也用于集合内的搜索。如清单 11-63 所示，在 URL 中使用音乐类型 `genre` 作为参数，在 HTTP 响应中返回相关的类型。

清单 11-63 使用 `$in` 查询操作符筛选类型

```
// example-011/album-routes.js
```

```
/**
 * GET /album/genre/(genre)/related
 * @param req
 * @param cb
 */
routes.GET['^/album/genre/([a-zA-Z]+)/related$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var principalGenre = req.params[0];
  var criteria = {
    genre: {$in: [principalGenre]}
  };
  Album.find(criteria)
```

```

    .lean(true)
    .select('-_id genre')
    .exec(function (err, albums) {
      if (err) return cb(500, err);
      var relatedGenres = [];
      albums.forEach(function (album) {
        album.genre.forEach(function (albumGenre) {
          // don't include the principal genre
          if (albumGenre === principalGenre) return;
          // ensure duplicates are ignored
          if (relatedGenres.indexOf(albumGenre) < 0) {
            relatedGenres.push(albumGenre);
          }
        });
      });
      cb(200, {genre: principalGenre, related: relatedGenres});
    });
  });
};

```

```

example-011$ curl -X GET http://localhost:8080/album/genre/Dance/related
{"genre":"Dance","related":["Electronica","Soundtrack"]}

```

为了确定“相关”类型，使用条件对象选择在文档类型数组中有主要类型元素的专辑。然后将所有其他类型列表添加到已经分配给专辑列表的结果集，并返回给客户端。虽然 `Album.genre` 是一个数组，但 MongoDB 会将其转换为与 `$in` 运算符中元素相匹配的类型值。因为 `Query.select()` 方法仅处理 `route` 中包含的数据，所以 `Query.select()` 方法排除 `_id` 属性，仅包含类型属性。

`$in` 运算符对于在数组中查找标量值是非常有用的，但是搜索复杂对象需要另一种不同的方法。例如，`Album.tracks` 中的每个子文档均有自己的属性和值。为了搜索专辑中符合条件的曲目，可以通过完整的属性路径获取曲目的属性。清单 11-64 中，将获取曲目含有与条件对象相匹配的 `tracks.title` 属性的专辑。

清单 11-64 在条件对象中使用子文档路径

```

// example-012/album-routes.js
/**
 * GET /album(?composer={string}&title={string}&track={string})
 * @param req
 * @param cb
 */
routes.GET['^/album(?:\?.+)?$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var criteria = {};
  // ...
  if (req.query.track) {
    criteria['tracks.title'] = req.query.track;
  }
  // ...
  Album.find(criteria)
    .lean(true)
    .exec(function (err, albums) {
      if (err) return cb(500, err);
      cb(200, albums);
    });
};

```

```

example-012$ curl -X GET http://localhost:8080/album?track=The%20Looking%20Glass
[{"_id":"54ed1dcb6fb525ba25529bd1","composer":"Kerry Muzzey","title":"Renaissance"... }

```

3. \$and 与 \$or 运算符

简单条件对象可以通过使用普通对象进行属性查询。例如，使用清单 11-65 中的简单条件对象，即可查找已发行的专辑。

清单 11-65 简单条件对象

```
Album.find({inPublication: true}, function (err, albums) { /*...*/ });
```

但是，简单条件对象方法无法处理复杂的复合查询，如清单 11-66 所示的查询语句。

清单 11-66 复杂的查询语句

```
(select albums that
  (
    (are in publication and were released within the last two years) or
    (are categorized as classical and priced between $9 and $10)
  )
)
```

幸运的是，\$and 与 \$or 运算符可用于构建查找所需专辑的条件对象。\$and 与 \$or 运算符均可接收条件对象数组作为运算数，包括简单查询语句或是由 \$and、\$or 等其他合法的查询操作符组成的复杂查询语句。\$and 运算符对条件对象数组中的元素执行逻辑与（AND）操作，仅选择一个匹配所有条件对象的文档。相反，\$or 运算符执行逻辑或（OR）操作，选择所有满足任意条件对象的文档。

清单 11-67 中，使用两种复合运算符组成的条件对象查找专辑。需要注意的是，条件对象中的关键字是属性名，而复合条件对象中的关键字是由复合运算符组成的简单和/或复杂的条件对象。

清单 11-67 使用 \$and 与 \$or 运算符查找所推荐的专辑

```
// example-012/album-routes.js
/**
 * GET /album/recommended
 * @param req
 * @param cb
 */
routes.GET['/^\\/album\\/recommended$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var nowMS = Date.now();
  var twoYearsMS = (365 * 24 * 60 * 60 * 1000 * 2);
  var twoYearsAgo = new Date(nowMS - twoYearsMS);

  var criteria = {
    $or: [
      // match all of these conditions...
      { $and: [{inPublication: true}, {releaseDate: {$gt: twoYearsAgo}}] },
      // OR
      // match all of these conditions...
      { $and: [{genre: {$in: ['Classical']}}, {price: {$gte: 5, $lte: 10}}] }
    ]
  };

  Album.find(criteria)
    .lean(true)
    .select('-_id -tracks')
    .exec(function (err, albums) {
```



```

    if (err) return cb(500, err);
    cb(200, albums);
  });
};

example-012$ curl -X GET http://localhost:8080/album/recommended
[{"composer": "Kerry Muzzey", "title": "Renaissance", "price": 4.95... },
{"composer": "Audiomachine", "title": "Tree of Life", "price": 9.49... },
{"composer": "Jessica Curry", "title": "Dear Esther", "price": 6.99... }]

```

4. \$regex 运算符

通常情况下，查找与精确文本字段相匹配的文档得到的并非是最优结果。正则表达式可用于扩展搜索，查找文档的多个字段与一个特定的查询参数相类似的结果。对于基于 SQL 的语言，like 运算符可用于这个目的，但是 MongoDB 支持正则表达式。**\$regex** 运算符为条件对象属性添加正则表达式，搜索与正则表达式匹配的文档。它通常与**\$options** 运算符配合使用，包括有效的正则表达式标记，如 i (区分大小写)。清单 11-68 中，使用查询参数 **owner**，并转换为正则表达式，对每个文档的 **owner** 属性进行计算。

清单 11-68 使用正则表达式查找

```

// example-012/library-routes.js
/**
 * GET /library?
 * @param req
 * @param cb
 */
routes.GET['^/library(?:\?.+)?$'] = function (req, cb) {
  cb = httpd.asJSON(cb);
  var criteria = {};
  if (req.query.owner) {
    criteria.owner = {
      $regex: '^.*' + req.query.owner + '.*$',
      $options: 'i'
    }
  } else {
    return cb(404, {message: 'please specify an owner'});
  }
  Library.find(criteria)
    .populate('albums')
    .exec(function (err, libraries) {
      if (err) return cb(500, err);
      cb(200, libraries);
    });
};

```

条件对象指定了使用正则表达式的目标属性以及一个对象。该对象同时包括正则表达式 (**\$regex** 属性) 和匹配选项 (**\$option** 属性)。清单 11-69 中，curl 命令使用 **owner=cloud** 作为查询参数。由于清单 11-68 中的正则表达式使用通配符，并且正则表达式选项 **i** 指定区分大小写，该规则将仅返回 Nicholas Cloud 的一条 MongoDB 记录。清单 11-69 显示了 curl 命令以及 HTTP 响应的输出结果。

清单 11-69 通过 curl 的 owner 参数查找

```

curl -X GET http://localhost:8080/library?owner=cloud
[{"_id": "54ed249312c06b3726d3abcd", "owner": "Nicholas Cloud"...}]

```

5. 高级查询运算符

还有许多可用于 Mongoose 查询的 MongoDB 运算符，需要更多篇幅深入分析每个运算符。表格 11-3 概述了高级查询运算符。

表 11-3

附加高级查询操作符

操作符	描述
\$not, \$nor	负逻辑运算符，可连接查询语句，并选择相匹配的文档
\$exists	选择指定的属性存在的文档（MongoDB 文档在技术上是无模式的）
\$type	选择指定属性是给定的类型的文档
\$mod	选择指定字段的模运算符能返回指定结果的文档中（如选择所有价格可被 3.00 整除的专辑）
\$all	选择包含所有指定属性的文档
\$size	选择给定属性与给定大小一致的文档
\$elemMatch	选择子文档匹配多个条件的文档

11.6 小结

MongoDB 是无模式的，在设计上非常灵活，但是应用开发者们常常在应用程序代码中添加数据约束来执行业务规则，以确保数据的完整性，满足现有的应用程序抽象，或实现诸多其他目。Mongoose 承认并接受这一事实，以便方便快捷地处理应用程序与数据存储之间的通信。

Mongoose 模式为自由形式的数据添加约束。Mongoose 模式定义数据存储的形式（模式）和有效性，增强数据约束，在文档之间创建关联，并通过中间件提供文档生命周期。

模型提供了完整、可扩展的查询接口。条件对象与 MongoDB 查询语法一致，可用于查找特定数据。可链式调用的查询方法可用于控制属性选择、引用关联、甚至完整文档或原生 JSON 对象。自定义静态方法可用于封装复杂的条件对象，并且可为模型添加更多查询，保持应用程序之间的数据处理互不干扰。

最后，Mongoose 文档可以通过包含域逻辑的自定义实例方法，自定义 getter 与 setter 方法，有助于进一步操作属性。

Knex 和 Bookshelf

说我即将步入死神之门的报道完全是夸大其词。

——萨缪尔·兰亨·克莱门（马克·吐温）

在这一章中，我们会研究两个库，它们互相配合，可以帮助解决 Node.js 开发者在处理关系型数据库时经常遇到的问题。第一个是 Knex，它提供了灵活一致的接口和一些知名的 SQL 平台交互，如 MySQL 和 PostgreSQL。第二个是 Bookshelf，它在 Knex 的基础上开发，给开发者提供了一个强大的对象关系映射（ORM）库。它能简化为实体（实体构成应用的数据模式）建模的过程，以及简化实体之间错综复杂的关系。熟悉 Backbone.js 的且知道 Backbone.js 强调用 Model 和 Collection 来模式化数据的读者，会发现 Bookshelf 会有种亲切感，因为这个库和 Backbone.js 遵循相同的模式，并且提供了很多相同的 API。

在这一章中，你将学会做以下事情：

- 用 Knex 查询构建器创建 SQL 查询
- 通过 promise 的帮助，无需内嵌的回调函数就能进行复杂的数据库交互
- 通过使用事务（transaction）保证应用数据的完整性
- 通过 Knex 迁移脚本管理对数据库模式（schema）的修改
- 通过 Knex 种子（seed）脚本给数据库填充样例数据
- 定义 Bookshelf 模型间一对一、一对多和多对多的关系
- 采用渴望加载（eager loading）方式高效获取基于 Bookshelf 关系的复杂对象图谱（object graph）

注意 本章中大部分的例子都会大量使用 Bookshelf 和 Knex 提供的 promise 化和 Underscore 化的 API。所以，对这两个概念不熟悉的读者，建议首先阅读第 14 章（关于 Q）和第 16 章（关于 Underscore 和 Lo dash）。

12.1 Knex

Knex 提供了一个数据库抽象层（DBAL），为 MySQL、PostgreSQL、MariaDB 和 SQLite3 提供了一套统一的接口，开发者在使用这些 SQL 数据库的时候无须再考虑不同数据库平台之间语法上和响应格式之间的微小差异。基于这些数据库平台开发的应用结合使用 Knex 之后能受益良多，包括以

下几点:

- promise 化的接口设计使得对异步过程进行更清晰的控制
- 支持流的接口能在应用需要时高效地传输数据
- 可以使用统一接口给不同的数据库平台创建查询和数据库模式
- 支持事务

除了库本身, Knex 还提供了命令行工具。开发者可以利用命令行工具做以下事情:

- 创建、实施和(有必要的话)数据库迁移和脚本化的数据库模式更改, 这些过程都可以结合到应用的源代码中。
- 创建数据库“种子”脚本, 它能为应用数据库填充样例数据以供本地开发和测试。所有这些主题都会在本章进行详细的讲解。

12.1.1 安装命令行工具

在继续之前, 你首先得确保安装了 Knex 命令行工具。Knex 命令行工具在 npm 上能获取到, 具体安装过程如清单 12-1 所示。

清单 12-1 通过 npm 安装 Knex 命令行工具

```
$ npm install -g knex $ knex --version
Knex CLI version: 0.7.3
```

12.1.2 把 Knex 添加到你的项目

除了安装 Knex 命令行工具, 你还需要把 Knex 的 npm 模块包含进每个项目作为项目的本地依赖, 因为项目中需要用到 Knex, 另外也需要依赖数据库模块, 如清单 12-2 所示。

清单 12-2 通过 npm 安装 Knex 模块和数据库模块作为本地项目依赖

```
$ npm install knex --save
# Supported database libraries include (be sure to --save):
$ npm install mysql
$ npm install mariasql
$ npm install pg
$ npm install sqlite3
```

■ **注意** SQLite 是一个自包含、无需服务端的数据库。它把数据包含在磁盘上的单个文件中, 从而无须再依赖别的工具。如果你刚好没有权限访问像 MySQL 这样的数据库, sqlite3 库可以给你提供一个快速简便、无需额外安装的途径来试验 Knex 的功能。本章中所有用到的例子都会使用这个库。

12.1.3 配置 Knex

当所有依赖都安装好之后, 接下来就是要在项目中初始化 Knex。清单 12-3 展示了使用 MySQL、PostgreSQL 和 MariaDB 时代码的写法, 清单 12-4 则展示了使用 SQLite3 时该如何初始化 Knex。

清单 12-3 使用 MySQL、PostgreSQL 和 MariaDB 时初始化 Knex（如果需要的话，把 `mysql` 替换成 `pg` 或 `mysql`）

```
var knex = require('knex')({
  'client': 'mysql',
  'connection': {
    'host': '127.0.0.1',
    'user': 'user',
    'password': 'password',
    'database': 'database'
  },
  'debug': false // 调试时将该属性设置成 true
});
```

清单 12-4 使用 SQLite3 时初始化 Knex
// `example-sqlite-starter/lib/db.js`

```
var knex = require('knex')({
  'client': 'sqlite3',
  'connection': {
    'filename': 'db.sqlite'
  }
});
```

从上面的对比可以看出，SQLite3 的配置相比其他数据库的配置更加简洁。我们只需要提供一个文件名（`db.sqlite`），SQLite 会把数据都存储在这个文件中，而不需要提供连接相关的设置。

12.1.4 SQL 查询构建器

Knex 的主要关注点在于为开发者提供统一的接口和多种 SQL 数据库交互，而无须担心不同数据库之间语法和返回格式上的细微差异。为了实现这个目标，Knex 提供了很多方法，大部分方法可以归为两类：查询构建器方法和接口方法。

1. 查询构建器方法

查询构建器方法是指那些帮助开发者创建 SQL 查询的方法，如 `select()`、`from()`、`where()`、`limit()` 和 `groupBy()`。据最新统计，Knex 共提供了 40 多种这样的方法来创建跨平台的查询。清单 12-5 列出了一个简单的 SQL 查询，以及用一个例子来验证如何用 Knex 创建这样的查询。

清单 12-5 用 Knex 创建简单的 SQL 查询

```
// example-sqlite-starter/example1.js
// SELECT id, name, postal_code FROM cities;knex.select('id', 'name', 'postal_code').
from('cities');
```

清单 12-5 中的例子验证了用 Knex 创建 SQL 查询的基本方法，但是并没有给出这个库的全貌。随着我们继续了解多种多样的接口方法，它将变得越来越清晰。正是因为有这些方法，我们才能开始提交我们的查询并处理返回的结果。

2. 接口方法

Knex 提供了很多接口方法，让我们用多种简便的方式来提交和处理我们的查询。在这一部分，我们会看众多接口方法中最有用的两种方法。

(1) Promise

JavaScript 的事件驱动的本质决定它特别适合用来高效处理复杂的异步任务。JavaScript 开发者管理异步控制流的传统方式是通过采用回调函数，如清单 12-6 所示。

清单 12-6 简单的回调函数

```
var request = require('request');
request({
  'url': 'http://mysite.com',
  'method': 'GET'
}, function(err, response) {
  if (err) throw new Error(err);
  console.log(response);
});
```

回调函数让我们能推迟特定部分代码的执行时间到合适的时间，这样的方法易于理解和实施。然而不幸的是，当应用变得越来越复杂之后，管理这些函数也会变得非常困难。设想这样一种情景：我们需要在清单 12-6 中的第一个响应返回之后加一个额外的异步过程。为了实现这个目标，我们需要再增加额外的、内嵌的回调函数。随着越来越多这样的额外的异步过程被添加到代码里，我们会开始感受到很多程序员所说的“回调地狱”（callback hell）和“末日金字塔”（pyramids of doom），他们用这些词来形容这种回调方式导致的不可维护的“意大利面条式”的代码。

值得庆幸的是，JavaScript 中的 promise 的概念给开发者提供了一种简便的方法来解决这种问题。Knex 也大量采用这种方式来解决类似问题，它用来提交和处理查询的接口都是基于 promise 的。清单 12-7 展现了这样的 API。

清单 12-7 Knex 提供的基于 promise 的 API

```
// example-sqlite-starter/example2.js
```

```
knex.pluck('id').from('cities').where('state_id', '=', 1)
  .then(function(cityIds) {
    return knex.select('id', 'first_name', 'last_name').from('users')
      .whereIn('city_id', cityIds);
  })
  .then(function(users) {
    return [
      users,
      knex.select('*').from('bookmarks').whereIn('user_id', _.pluck(users, 'id'))
    ];
  })
  .spread(function(users, bookmarks) {
    _.each(users, function(user) {
      user.bookmarks = _.filter(bookmarks, function(bookmark) {
        return bookmark.user_id = user.id;
      });
    });
    console.log(JSON.stringify(users, null, 4));
  })
  .catch(function(err) {
    console.log(err);
  });
```

在这个例子中，三个查询依次被提交：

1. 查询特定州下的城市
2. 查询 1 中城市下的用户
3. 查询 2 中每个用户的书签

当最后依次查询返回之后，我们会把每个书签归给合适的用户并展现结果，如清单 12-8 所示。

清单 12-8 清单 12-7 中的代码执行后记录到 Console 中的数据

```
[
  {
    "id": 1,
    "first_name": "Steve",
    "last_name": "Taylor",
    "bookmarks": [
      {
        "id": 1,
        "url": "http://reddit.com",
        "label": "Reddit",
        "user_id": 1,
        "created_at": "2015-03-12 12:09:35"
      },
      {
        "id": 2,
        "url": "http://www.theverge.com",
        "label": "The Verge",
        "user_id": 1,
        "created_at": "2015-03-12 12:09:35"
      }
    ]
  }
]
```

因为 Knex 提供了基于 promise 的接口，我们的代码最多只有一层缩进，从而保证我们的应用代码易于理解。更重要的是，当这个过程中任何一个步骤发生错误时，错误会很方便地被捕捉，并且被最后的 catch 语句处理。

注意 JavaScript 的 promise 是一个特别强大的工具，它能让开发者以易于理解和维护的方式编写复杂的异步代码。如果你对这个概念不太熟悉，那么建议先跳到第 14 章阅读关于 promise 库 Q 的内容，从而对 promise 形成更深入的理解。

(2) 流

用 Node.js 编写应用代码的好处之一是 Node.js 能以非常高效的方式执行 I/O 密集型的过程。不像 PHP、Python 和 Ruby 这些同步语言，Node.js 能仅在一个线程里就处理数千个并发请求。这能让开发者以最少的资源写出能满足巨大需求的应用。Node.js 提供了一系列重要的工具来实现它的目标，其中很重要的一个工具就是流。

在讲流之前，我们先看一下清单 12-9 中一个传统的 JavaScript 回调函数的例子。

清单 12-9 JavaScript 回调函数接受加载的文件的内容

```
var fs = require('fs');
fs.readFile('data.txt', 'utf8', function(err, data) {
  if (err) throw new Error(err);
  console.log(data);
});
```

在这个例子中，我们使用 Node.js 的 fs 库中的 readFile() 方法来读取一个文件的内容。当数据被完全加载到内存中之后，它就会被传给回调函数来做进一步处理。这个过程很简单并且易于理解。但是，这种过程不是很高效，因为应用需要首先把整个文件的内容全部加载到内存中，然后传

回给我们。如果是小文件的话，这不是大问题；但如果是大文件的话，就可能导致问题，是否会导致问题取决于运行当前应用的服务器的可用资源大小。

Node.js 的流通过把数据切分成多个小的数据块来传输，就能解决上面的问题。这样做就能避免开发者把大量的服务器资源消耗在单个请求上。清单 12-10 中的例子完成了和上一个例子一样的功能，但是无须再将整个文件一次性加载到内存中。

清单 12-10 利用一对共同协作的 Node.js 流，高效加载和展示一个文件的内容

```
// example-read-file-stream/index.js
```

```
var fs = require('fs');
var Writable = require('stream').Writable;
var stream = fs.createReadStream('data.txt');
var out = Writable();
out._write = function(chunk, enc, next) {
  console.log(chunk.toString());
  next();
};
stream.pipe(out);
```

流是 Node.js 中一个被用得相对不多的功能。这很不幸，因为流恰好就是 Node.js 中最强大的功能之一。值得庆幸的是，Knex 提供了流式的接口来接收请求，这使得我们能利用流的优点，如清单 12-11 所示。

清单 12-11 利用 Knex 提供的流接口，处理请求的结果

```
var Writable = require('stream').Writable;
var ws = Writable();
ws._write = function(chunk, enc, next) {
  console.dir(chunk);
  next();
};
var stream = knex.select('*').from('users').stream();
stream.pipe(ws);
```

在这个例子中，对 `users` 表（对某些应用来说可能非常大）的查询结果以较小的数据块的形式传给可写流，而不是完整地进行传送。这个过程也可以结合 `promise` 接口，从而构建更加健壮的应用，如清单 12-12 所示。

清单 12-12 把 Knex 提供的流和 `promise` 接口结合起来，保证更好的错误处理

```
var Writable = require('stream').Writable;
var ws = Writable();
ws._write = function(chunk, enc, next) {
  console.dir(chunk);
  next();
};
knex.select('*').from('users').stream(function(stream) {
  stream.pipe(ws);
}).then(function() {
  console.log('Done.');
```

在这个例子中，我们把 Knex 提供的流和 `promise` 接口结合起来了。当回调函数传递给 `stream()` 方法时，这个回调函数会接收前面生成的 `promise` 对象，而不是像上个例子那样直接被返回。同时，

stream()方法也返回一个 promise 对象。当流传输完毕时，它就会被解决。

■ **注意** Knex 提供的流接口目前和 MySQL、PostgreSQL、MariaDB 兼容，SQLite3 尚不支持。

(3) 事务

使用 ACID 兼容的关系型数据库的好处之一在于，它们能将多个查询组合成一个工作单元（事务）。它们要么一起成功，要么一起失败。换句话说，如果事务中的任何一个查询失败，所有先前的查询所做的改变都会被撤销。

举个例子，设想一下你的银行账户里发生了一笔金融交易。假如你想在表妹的生日那天给她转 25 美元，这些钱首先得从你的名下取出，然后转到表妹的名下。设想这样的情景：交易系统因为某种原因（例如，由一行错误代码或者一次较大的系统错误导致）突然崩溃，而且崩溃恰好发生在钱从你账户取出但还没有转到表妹账号名下之前。如果没有事务所保证的安全机制，这些钱实际上就等于凭空消失了。而有了事务，开发者就能确保这样的过程只能“完整”地发生，永远不会使数据处于不一致的状态。

■ **注意** 缩写词 ACID（原子性、一致性、隔离性、持久性）是代表数据库事务的一组性质。原子性表示事务要么完整地成功，要么完整地失败，这样的事务被称作“原子的”。

本章中之前的例子展示了使用 Knex 创建和发送数据库查询的过程。在继续之前，我们先看一个没有利用事务的例子。然后我们会转而让这个例子使用事务来实现。

如清单 12-13 所示，moveFunds() 声明当它被调用时，使用 knex 对象把一个账户里的钱转移到另一个。这个函数会返回一个 promise，当过程完成时，promise 要么被解决，要么被拒绝，视调用成功或失败而定。然而例子中有一个严重的错误，你能发现吗？

清单 12-13 moveFunds() 演示了在没有事务安全性保证的前提下，把一个账户的钱转移到另一个账户

```
// example-financial/bad.js
```

```
/**
 * 将特定数量的钱从 sourceAccountID 账户转移到 destAccountID 账户
 */
var moveFunds = function(sourceAccountID, destAccountID, amount) {
  return knex.select('funds').from('accounts')
    .where('id', sourceAccountID)
    .first(function(result) {
      if (!result) {
        throw new Error('Unable to locate funds for source account');
      }
      if (result.funds < amount) {
        throw new Error('Not enough funds are available in account');
      }
      return knex('accounts').where('id', sourceAccountID).update({
        'funds': result.funds - amount
      });
    }).then(function() {
      return knex.select('funds').from('accounts')
        .where('id', destAccountID);
    }).first(function(result) {
      if (!result) {
        throw new Error('Unable to locate funds for destination account');
      }
    })
}
```

```

    return knex('accounts').where('id', destAccountID).update({
      'funds': result.funds + amount
    });
  });
};

/* 将 25 美元从账户 1 转移到账户 2 */
moveFunds(1, 2, 25).then(function(result) {
  console.log('Transaction succeeded.', result);
}).catch(function(err) {
  console.log('Transaction failed!', err);
});

```

在这个例子中，需要经过以下步骤才能实现将源账户的钱转移到目标账户：

1. 首先要知道源账户里的可用资金量。
2. 如果可用资金不够，则抛出错误。
3. 要转移的钱从源账户扣除。
4. 知道目标账户里目前的可用资金量。
5. 如果找不到目标账户，则抛出错误。
6. 要转移的钱被加到目标账户。

如果你还是没有发现错误的话，那么我告诉你错误在第 5 步。假如说目标账户没找到，错误被抛出，但是此时钱已经从源账户里扣除，我们可以有很多种办法解决这个问题。我们可以捕捉到这个错误，并把钱加回到源账户。但是即便如此，我们仍然不能保证不发生未知的错误，如由于网络错误或者在这个过程正在进行的时候应用服务器由于掉电突然就完全崩溃了。

在这个时候，事务的作用就体现出来了。在清单 12-14 中，`moveFunds()` 被重构成将整个过程包含在单个“原子的”事务操作中，要么完整地成功，要么完整地失败。注意 `trx` 对象，事务相关的查询都是由此构建的。

清单 12-14 清单 12-13 中的过程采用事务操作实现

// example-financial/index.js

```

/**
 * 将特定数量的钱从 sourceAccountID 账户转移到 destAccountID 账户
 */
var moveFunds = function(sourceAccountID, destAccountID, amount) {
  return knex.transaction(function(trx) {
    return trx.first('funds')
      .from('accounts')
      .where('id', sourceAccountID)
      .then(function(result) {
        if (!result) {
          throw new Error('Unable to locate funds for source account');
        }
        if (result.funds < amount) {
          throw new Error('Not enough funds are available in account');
        }
        return trx('accounts').where('id', sourceAccountID)
          .update({
            'funds': result.funds - amount
          });
      })
      .then(function() {

```

```

        return trx.first('funds')
            .from('accounts')
            .where('id', destAccountID);
    })
    .then(function(result) {
        if (!result) {
            throw new Error('Unable to locate funds for destination account');
        }
        return trx('accounts').where('id', destAccountID)
            .update({
                'funds': result.funds + amount
            });
    });
});
});

/* 将 25 美元从账户 1 转移到账户 2 */
displayAccounts()
    .then(function() {
        return moveFunds(1, 2, 25);
    }).then(function() {
        console.log('Transaction succeeded.');
```

可以看出, 清单 12-14 中使用事务的例子和清单 12-13 中的例子非常相像, 只是在很关键的地方不同。清单 12-14 中并不是直接在 `knex` 对象上调用函数来构建查询, 而是首先通过调用 `knex.transaction()` 初始化了一个事务对象。我们给 `knex.transaction()` 提供回调函数, 然后可以使用第一个参数 (`trx`) 构建一系列的查询。从这个时刻开始, 任何通过 `trx` 创建的查询都会一起成功或一起失败。`knex.transaction()` 会返回 `promise`。当事务完成时, 它要么被解决, 要么被拒绝, 然后就能把事务这个过程整合到更大的基于 `promise` 的操作中。

12.1.5 迁移脚本

就像应用的源代码注定会不停地更改一样, 存储的模式也会不时地需要改变。当这样的改变发生时, 保证这些改变能被复制、共享、在有必要的时候被撤销以及被追踪记录是相当重要的。数据库迁移脚本能给开发者提供一种简便的模式来实现这些目标。

Knex 迁移脚本由两个函数组成: `up` 和 `down`, 如清单 12-15 所示。脚本中的 `up` 函数负责以一种期望的方式来修改数据库的模式 (如创建表、增加列), 而 `down` 函数负责恢复数据库模式到之前的状态。

清单 12-15 Knex 迁移脚本: 用 `up` 函数创建新表, 用 `down` 函数删除表
 // example-sqlite-starter/migrations/20150311082640_states.js

```

exports.up = function(knex, Promise) {
    return knex.schema.createTable('states', function(table) {
        table.increments().unsigned().primary().notNullable();
        table.string('name').notNullable();
        table.timestamp('created_at').defaultTo(knex.fn.now()).notNullable();
    });
};
exports.down = function(knex, Promise) {
```

```
    return knex.schema.dropTable('states');
  };
}
```

1. 项目迁移配置

Knex 的命令行工具给开发者提供了一种简便的方式来创建和管理迁移脚本。开始之前，你首先需要在项目根目录下执行下面的命令来新建一个特殊的配置文件。

```
$ knex init
```

运行这个命令之后，一个叫 `knexfile.js` 的文件会被创建，里面包含的内容和清单 12-16 中的类似。有必要的話，你应该修改这个文件中的内容。每当 Knex 迁移脚本运行时，Knex 会根据这个文件的内容和 `NODE_ENVIRONMENT` 环境变量来决定连接设置。

■ **注意** 在 OSX 和 Linux 系统上，在终端可以通过执行 `export ENVIRONMENT_VARIABLE= value` 来设置环境变量。而在 Windows 系统上，则可以执行 `set ENVIRONMENT_VARIABLE=value`。

清单 12-16 `knexfile.js`

```
// example-sqlite-starter/knexfile.js
```

```
module.exports = {
```

```
  'development': {
    'client': 'sqlite3',
    'connection': {
      'filename': './db.sqlite'
    },
    'seeds': {
      'directory': './seeds'
    }
  },
```

```
  'staging': {
    'client': 'postgresql',
    'connection': {
      'database': 'my_db',
      'user': 'username',
      'password': 'password'
    },
    'pool': {
      'min': 2,
      'max': 10
    }
  }
};
```

2. 创建迁移脚本

我们已经有了 Knex 配置文件，接下来就要创建第一个迁移脚本了。创建脚本的命令如下。

```
$ knex migrate:make users_table
```

当你创建自己的迁移脚本时，记得把命令中 `users_table` 部分替换成实际迁移中应该的内容。执行完这个命令之后，Knex 会为你创建一个迁移脚本，文件内容和清单 12-17 中的代码类似。

清单 12-17 新建的 Knex 迁移脚本

```
exports.up = function(knex, Promise) {
};

exports.down = function(knex, Promise) {
};
```

当你创建迁移脚本之后，你项目的文件模式应该类似清单 12-18 所示。

清单 12-18 创建迁移脚本之后的项目文件模式

```
├── knexfile.js
└── migrations
    └── 20141203074309_users_table.js
```

■ **注意** Knex 的迁移脚本保存在项目根目录下的 `migrations` 文件夹下。如果这个文件夹不存在，Knex 会帮你创建一个。Knex 会自动在迁移脚本文件名之前加上时间戳，如清单 12-18 所示。这保证了这些项目迁移脚本总是能按创建的顺序排列。

现在就轮到我们在新创建的迁移脚本里修改 `up` 和 `down` 函数了。让我们来看看两种不同的方法。

(1) 通过模式构建器函数来定义模式更新

除了提供构建查询的方法之外，Knex 还提供方法用来定义数据库的内在模式（模式）。通过这些“模式构建器”方法，开发者可以创建跨平台的“蓝图”来描述表格、列、索引和关系等构成数据库的元素。这些“蓝图”然后就可以被应用到支持的平台下，从而生成期望的数据库。清单 12-15 所示的迁移脚本展示了 Knex 模式构建器，而清单 12-19 则展示了脚本中 `up` 函数所生成的查询。

清单 12-19 使用清单 12-15 中的模式构建器方法创建的 SQL 查询

```
// example-raw-migration/migrations/20150312083058_states.js
```

```
CREATE TABLE states (
  id integer PRIMARY KEY AUTOINCREMENT NOT NULL,
  name varchar(255) NOT NULL,
  created_at datetime NOT NULL DEFAULT(CURRENT_TIMESTAMP)
);
```

模式构建器方法会非常有用，它们可以让开发者很方便地定义模式，并且能把这些模式应用到任何一个 Knex 支持的平台。同时，它们只需要开发者知道很少的原生 SQL 查询相关的知识，从而让那些对直接操作 SQL 数据库没有太多经验的开发者也能很快应用起来。但是这也意味着，模式构建器函数也会有局限性。为了能提供一个通用的接口来定义出在多个平台都能使用的模式，Knex 必须做确定的决定，这可能会让你觉得不太适应。对直接操作 SQL 数据库有更多经验的开发者可能会希望完全绕过模式构建器函数，从而直接构建他们的 SQL 查询。这很好实现，待会儿我们就能看到。

(2) 通过原生 SQL 查询定义模式更新

在清单 12-20 中，我们可以看到 Knex 迁移脚本通过原生 SQL 查询创建了一个 `users` 表，这是通过使用 `knex.schema.raw()` 方法来完成的。当被调用时，这个方法会返回一个 `promise`，它会根据查询的成功或失败被解决或被拒绝。

清单 12-20 用原生 SQL 查询来定义 Knex 迁移脚本

```
// example-raw-migration/migrations/20150312083058_states.js

var multiline = require('multiline');
```

```

exports.up = function(knex, Promise) {
  var sql = multiline.stripIndent(function() { /*
    CREATE TABLE states (
      id integer PRIMARY KEY AUTOINCREMENT NOT NULL,
      name varchar(255) NOT NULL,
      created_at datetime NOT NULL DEFAULT(CURRENT_TIMESTAMP)
    );
  */});
  return knex.schema.raw(sql);
};

exports.down = function(knex, Promise) {
  return knex.schema.raw('DROP TABLE states;');
};

```

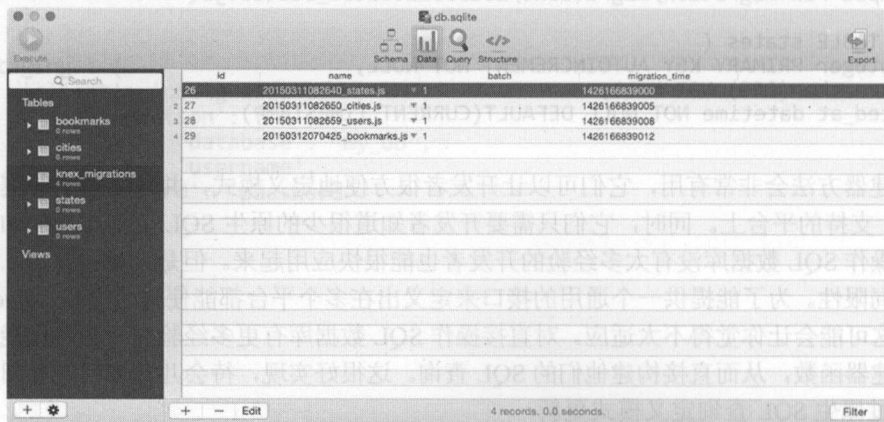
■ **注意** 清单 12-20 中的例子用了一个和 Knex 无关的额外的库：**multiline**。**multiline** 库非常有用是因为它可以用来定义一大段跨越多行的文本，并且无须在每行之后增加连接的字符。

3. 执行 Knex 迁移

现在有了新建的迁移脚本，并且也达到执行状态，所以我们剩下的任务就是执行迁移，让数据能发生我们所期望的更新。执行的更新的命令如下。

```
$ knex migrate:latest
```

这个命令会让 Knex 执行所有还没被执行的迁移脚本，以脚本创建的顺序来执行。一旦完成之后，数据库就会发生我们所期望的改变。如果你好奇 Knex 是如何追踪哪些脚本已经被执行、哪些没有的，答案在 **knex_migrations** 这个 Knex 为自己创建的表里（见图 12-1）。在这个表里，Knex 维护了一个不断变动的列表，用来记录实施过的迁移变动。这个表的名字可以通过对配置文件（由 **knex init** 创建）的更改来更改。



	id	name	batch	migration_time
1	26	20150311082640_states.js	1	1426166839000
2	27	20150311082650_cities.js	1	1426166839005
3	28	20150311082659_users.js	1	1426166839008
4	29	20150312070425_bookmarks.js	1	1426166839012

图 12-1 **knex_migrations** 表：Knex 用来记录已经对数据库执行过的迁移脚本

4. 撤销 Knex 迁移

Knex 执行迁移脚本的行为并不是单向的，它们也可以被撤销，这在开发的时候尤其重要。命令如下：

```
$ knex migrate:rollback
```

这个命令会让 Knex 撤销最近一次 **knex migrate:latest** 执行所带来的更改。为了验证当前数

数据库所处的状态，可以执行下面的命令来查看数据库当前的迁移版本。

```
$ knex migrate:currentVersion
```

12.1.6 种子脚本

在之前的部分中，你学习了 Knex 的迁移脚本如何让你将对数据库模式的更改通过脚本实现，这个脚本可以分享给别人，可以在有必要的时候撤销操作，也可以在版本控制系统中记录。Knex 的种子脚本也有类似的目的，但是专注点在数据而非模式上。种子脚本提供了一种一致的方式来指定新建的数据库如何填充样例数据，从而创建一个新的开发环境并运行。清单 12-21 展示了本章的样例项目中的一个种子文件。

清单 12-21 简单的 Knex 种子脚本，用来移除 states 表中所有存在的记录并插入两条新的记录

```
// example-sqlite-starter/seeds/01-states.js
```

```
exports.seed = function(knex, Promise) {
```

```
  return Promise.join(
    knex('states').del(),
    knex('states').insert([
```

```
    {
      'id': 1,
      'name': 'Georgia'
```

```
    },
```

```
    {
      'id': 2,
      'name': 'Tennessee'
```

```
    ]
  );
```

```
});
```

1. 创建种子脚本

你可以通过下面的命令让 Knex 创建新的种子脚本。

```
$ knex seed:make users
```

Knex 会默认将新创建的种子脚本保存到项目根目录的 **seeds** 文件夹下。你可以通过修改项目的 **knexfile.js** 配置文件来自定义该文件夹（见清单 12-16）。

2. 执行种子脚本

为项目创建种子脚本之后，你可以通过执行下面的命令给数据库填充数据。

```
$ knex seed:run
```

注意 种子脚本总是按字母表顺序执行的。如果种子脚本的执行顺序很重要的话，起名字的时候就要注意，确保它们会按你希望的顺序执行。

12.2 Bookshelf

Bookshelf 是在 Knex 铺设的基础之上构建的。它提供了一个灵活的 ORM（对象关系映射）库，

简化了创建类（模型）来代表各种应用所需对象的过程。这个部分探索了多种使用 Bookshelf 的方法，以供开发者实现以下任务。

- 创建类（模型）来代表各种应用数据库中需要用到的表
- 给模型扩展出应用所需的特定的自定义行为
- 定义模型之间复杂的关系（一对一、一对多、多对多）
- 通过“渴求加载”（eager loading）的方式，方便地展示出模型之间的各种关系，而无须诉诸于复杂的 SQL 查询

熟悉 Backbone 的开发者会发现对 Bookshelf 也感到特别亲切，因为它们遵循了很多相同的模式并定义了很多相同的 API。你会很容易把 Bookshelf 描述成“服务器端的 Backbone”，然而这种描述和实际也并不会相差很多。

12.2.1 什么是对象映射关系

关系型数据库会以行的形式把信息存储在表中，每张表都用一列或多列来描述表中记录的各种属性——就像你用 Excel 表构建信息模式时一样。在多数应用中，我们通过创建不同的表，代表不同类型的实体（如“账户”、“用户”、“评论”）。而这些不同的实体之间的各种关系则是通过“外键”列实现的，如图 12-2 所示。

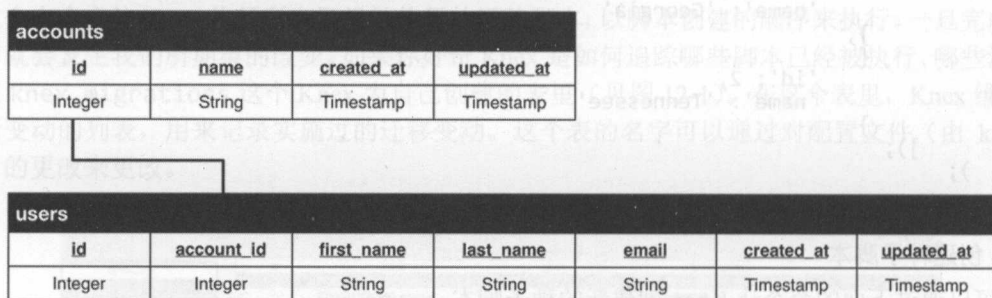


图 12-2 用户和账号（一个账户有一个或多个用户，用户属于账户）之间的关系
通过 users 表中的 account_id 外键列描述

用这种方式存储信息是非常强大的，而且是一般应用存储数据的主流方法，因为有很多好处（这些好处都超出了本书的范围）。不幸的是，这种方式 and 大多数应用倾向于查看数据时采用的面向对象的方式不太一致。

类似 Bookshelf 这样的对象关系映射（ORM）工具可以让开发者对关系型数据库中存储的扁平化的表信息，通过一系列互相连接的对象的形式进行交互。有了这样的工具之后，开发者就能进行交互和导航，从而达到一些期望的目标。从效果来说，ORM 的库给开发者提供了一个“虚拟的对象数据库”，让他们能更容易地对关系型数据库表中的扁平化记录进行交互。

12.2.2 创建 Bookshelf 模型

一个 Bookshelf 模型可以被认为是一个类，这个类被初始化之后就代表数据库中的一条记录。Bookshelf 模型最简单的形式是充当数据容器，并提供内建函数用来获取和设置属性值（如列的值）

以及用来创建、更新和删除记录。我们很快就会发现，当我们把 Bookshelf 模型用自定义的函数扩展并定义模型之间存在的关系的时候，它会变得越来越实用。

Bookshelf 模型是通过 `bookshelf.Model.extend()` 方法定义的，如清单 12-22 所示。在这个简单的例子中，我们定义了 User 模型，它的记录会被持久化地存储在数据库的 `users` 表中。

清单 12-22 简单的 Bookshelf 模型，代表应用中的用户

```
// example-bookshelf1/lib/user.js

var knex = require('./db');
var bookshelf = require('bookshelf')(knex);

var User = bookshelf.Model.extend({
  'tableName': 'users',
  'idAttribute': 'id' // 表的主键，默认为'id'
});

module.exports = User;
```

1. 创建新的实例

在清单 12-23 中，一个 Users 模型的实例被创建出来，修改后，最后被存储到数据库中。

清单 12-23 把 Users 的新实例存储到数据库中

```
// example-bookshelf1/create.js

var User = require('./lib/user');
var user = new User();

user.set({
  'first_name': 'Steve',
  'last name': 'Taylor',
  'email': 'steve.taylor@mydomain.com'
});

// Individual attributes can also be set as shown below
// user.set('first_name', 'Steve');

user.save().then(function(user) {
  // user has been saved
  console.log('User saved', user.toJSON());
  /*
  {
    first_name: 'Steve',
    last name: 'Taylor',
    email: 'steve.taylor@mydomain.com',
    id: 1
  }
  */
});
```

Bookshelf 提供了简便的 `forge()` 方法来稍稍简化上面的例子，如清单 12-24 所示。这个方法本质上只是创建了一个 Users 的新实例并返回，从而能让我们省去使用 `new` 关键字的过程。

清单 12-24 通过 `forge()` 方法创建 User 模型的新实例

```
// example-bookshelf1/forge.js

User.forge({
```

```

    'id': 1,
    'first_name': 'John'
  }).fetch().then(function(user) {
    /* An object containing every attribute / value for
    this model can be retrieved via the 'toJSON' method. */
    console.log(user.toJSON());
  });

```

2. 获取实例

User 模型的实例也可以通过相似的方式获取。在清单 12-25 中，我们创建了一个 User 的新实例，传入的 id 属性的值设为 1。当 `fetch()` 被调用时，Bookshelf 会使用所有传入模型上的值构建出一个 SQL 查询来获取期望的记录。在这个例子中，构建出的查询是：

```
SELECT * FROM users WHERE 'id' = 1;
```

清单 12-25 从数据库中获取 User 模型的一个实例

// example-bookshelf1/fetch.js

```

User.where({
  'id': 1
}).fetch().then(function(user) {
  // Individual attributes get be retrieved with the get method
  // console.log('first name', user.get('first_name'));
  console.log(user.toJSON());
});

```

3. 删除实例

正如模型实例可以被创建保存一样，它们也可以通过 `destroy()` 方法删除，如清单 12-26 所示。

清单 12-26 删除 User 模型的一个实例

// example-bookshelf1/destroy.js

```

User.where({
  'id': 1
}).fetch().then(function(user) {
  return user.destroy();
}).then(function() {
  console.log('User destroyed.');
```

在这个例子中，`destroy` 被当作 `user` 的一个实例方法调用。其实，我们可以直接让 Bookshelf 删除记录，而不需要先获取这个实例，如清单 12-27 所示。

清单 12-27 让 Bookshelf 删除特定的记录

```

User.where({
  'id': 1
}).destroy().then(function() {
  console.log('User destroyed.');
```

4. 获取多个模型（集合）

除了通过 `fetch()` 方法获取单个模型实例，我们还可以通过 `fetchAll()` 方法获取多个实例，如

清单 12-28 所示。

清单 12-28 获取 first_name 是 John 的所有用户实例

// example-bookshelf1/fetch-collection.js

```
User.where({
  'last_name': 'Doe'
}).fetchAll().then(function(users) {
  console.log(JSON.stringify(users.toJSON(), null, 4));
  /*
  [{
    "id": 3,
    "first_name": "John",
    "last_name": "Doe",
    "email": "john.doe@mydomain.com"
  },
  {
    "id": 4,
    "first_name": "Jane",
    "last_name": "Doe",
    "email": "jane.doe@mydomain.com"
  }]
  */
});
```

这个例子中，调用 `fetchAll()` 方法会返回一个 `promise`，这个 `promise` 解决后会得到一个有多多个用户的模型集合。这个集合自带了很多内建的方法，专门用来对多个模型进行操作。因为 `Bookshelf` 遵循了 `Backbone` 的模式，大多数 `Backbone` 的集合中的方法在 `Bookshelf` 中同样适用。清单 12-29 展示了一些常用的应用场景。

清单 12-29 常用的 `Bookshelf` 集合的方法

```
/* 遍历集合 */
users.each(function(user, index) {
  console.log(user, index);
});

/* 过滤出满足特定要求的模型的数组 */
users = users.filter(function(user, index) {
  if (user.get('last_name') === 'Smith') return true;
});

/* 一个更简单的过滤模型的方法，不需要提供函数调用 */
users = users.where({
  'last_name': 'Smith'
});

/* 返回第一个符合特定标准的结果 */
var johnSmith = users.find(function(user) {
  if (user.get('last_name') === 'Smith') return true;
});

/* 返回一个包含每个用户名字的数组 */
var firstNames = users.pluck('first_name');
```

5. 扩展自定义行为

在最简单的应用场景下，`Bookshelf` 只是充当了数据库中记录的容器的角色，并且提供了一些内

建的方法读写属性和进行保存或删除操作。虽然这样已经很实用了，但只有当 Bookshelf 模型为了适应应用的需求而开始扩展出它们独有的行为时，它们才算充分发挥了潜力。

一个这样行为的例子展示在清单 12-30 中了。在例子中，我们对上个例子中的 User 模型增加了 `sendEmail()` 方法。这么做让我们在给注册用户发送邮件过程中的复杂度大大降低了。

清单 12-30 给 User 模型扩展一个发送邮件的方法

```
var Promise = require('bluebird');
var Handlebars = require('handlebars');

var User = bookshelf.Model.extend({
  'tableName': 'users',
  /**
   * 发送一封邮件给用户。要求提供一个 'options' 对象，其中包含 'subject' 和
   * 'message' 的值。这些值会被编译成 Handlebars 模板函数，给它传入用户属
   * 性之后，返回的结果会用来生成邮件消息的内容。
   */
  'sendEmail': function(options) {
    var self = this;
    return Promise.resolve().then(function() {
      var subject = Handlebars.compile(options.subject)(self.toJSON());
      var message = Handlebars.compile(options.message)(self.toJSON());
      // 在这里使用你选择的邮箱库，以及合适的连接设置。});
    });
  });

User.where({
  'id': 1
}).fetch().then(function(user) {
  return user.sendEmail({
    'subject': 'Welcome, {{first_name}}',
    'message': 'We are happy to have you on board, {{first_name}} {{last_name}}.'
  });
});
```

除了这些从 Backbone 那里继承来的方法，Bookshelf 集合页还提供了一些自己的方法。清单 12-31 展示了 `invokeThen()` 方法的用法。该方法能很方便地让我们给集合中的每个模型调用特定方法。

清单 12-31 给集合中的每个模型调用 `sendEmail()` 方法

```
// example-bookshelf1/invoke-then.js

User.where({
  'last_name': 'Doe'
}).fetchAll().then(function(users) {
  return users.invokeThen('sendEmail', {
    'subject': 'Congratulations on having such a great name, {{first_name}}.',
    'message': '{{first_name}} really is a great name. Seriously - way to go.'
  });
}).then(function(users) {
  console.log('%s users were complimented on their name.', users.length);
});
```

例子中展示的 `invokeThen()` 方法会返回它自己的 `promise`。只有当集合中所有的模型调用 `sendEmail()` 被解决之后，这个 `promise` 才会被解决。这种模式也为我们提供了一个简便的方法和多个模型同时进行交互。

6. 校验

熟悉 Backbone 的人会发现 Bookshelf 的事件系统非常眼熟。单单针对校验来说，我们感兴趣的

就是 Bookshelf 触发的 `saving` 和 `destroying` 事件。通过研究这些事件，我们可以发现 Bookshelf 模型可以定制特殊的行为，以根据一些期望的标准允许或拒绝这些行为。清单 12-32 的例子中，含有“hotmail.com”字样的邮箱地址的用户不会被保存到数据库中。

清单 12-32 展示了 Bookshelf 的事件系统，允许运用自定义的校验规则

// example-bookshelf1/lib/user.js

```
var User = bookshelf.Model.extend({
  'tableName': 'users',
  'initialize': function() {
    this.on('saving', this._validateSave);
  },
  '_validateSave': function() {
    var self = this;
    return Promise.resolve().then(function() {
      if (self.get('email').indexOf('hotmail.com') >= 0) {
        throw new Error('Hotmail email addresses are not allowed.');
```

为了阻止 `save` 或 `destroy` 调用成功，只要介入模型的 `saving` 或 `destroying` 事件，然后传入自定义校验函数的引用就可以了。如果抛出错误，则调用会被阻止。异步验证的话，只要通过 `promise` 也是可以实现的。清单 12-33 中，自定义的验证函数返回了 `promise`，该 `promise` 最后被拒绝了。

清单 12-33 自定义校验函数，返回一个 `promise`

// example-bookshelf1/validation.js

```
User.forge({
  'first_name': 'Jane',
  'last_name': 'Doe',
  'email': 'jane.doe@hotmail.com'
}).save().then(function() {
  console.log('Saved.');
```

7. 自定义导出过程

之前的例子展示了 `toJSON()` 方法的用法。该方法默认会返回一个对象，对象中包含调用该方法的模型（如果是在集合中，就是集合中的每一个模型）的所有属性/值。如果你想自定义该方法返回的数据，可以通过覆盖 `toJSON()` 方法来实现，如清单 12-34 所示。

清单 12-34 自定义 `toJSON()` 方法返回的数据

```
var User = bookshelf.Model.extend({
  'tableName': 'users',
  'toJSON': function() {
    var data = bookshelf.Model.prototype.toJSON.call(this);
    data.middle_name = 'Danger';
    return data;
  }
});
```

在这个例子所覆盖的 `toJSON()` 方法里，我们首先调用了原型链上的 `toJSON()` 方法，它会返回这个方法没被覆盖时原本应该返回的数据。然后我们剥离出那些想隐藏的数据，添加了我们额外的数据，最后再返回。

上面这种形式最常用的场景是在使用 `User` 模型的时候，因为模型中常常有敏感密码信息。通过模型的 `toJSON()` 方法就能自动将这些信息剥离（清单 12-34），从而避免我们无意中通过 API 请求将这些敏感信息泄露出去。

8. 定义类属性

我们在之前的例子里见过的 `Bookshelf` 的 `extend()` 方法能接受两个参数：

- 一个是包含实例属性的对象，它会继承给新创建的模型实例
- 另一个是包含类属性的对象，它会直接赋给模型类

本章中之前的例子展示了将实例属性赋给 `extend()` 方法的过程，但是我们还没看到过类属性的使用例子。清单 12-35 就展示了类属性的使用。

清单 12-35 在 `User` 模型上定义 `getRecent()` 类方法

```
// example-bookshelf1/lib/user.js

var User = bookshelf.Model.extend({
  'tableName': 'users'
}, {
  /**
   * Returns a collection containing users who have signed in
   * within the last 24 hours.
   */
  'getRecent': function() {
    return User.where('last_signin', '>=', knex.raw("date('now', '-1 day')")).fetch();
  }
});

// example-bookshelf1/static.js

User.getRecent().then(function(users) {
  console.log('%s users have signed in within the past 24 hours.', users.length);
  console.log(JSON.stringify(users.toJSON(), null, 4));
});
```

类属性给我们提供了一个方便定义针对模型的各种辅助函数的地方。在稍微复杂的例子里，`getRecent()` 方法会返回一个 `promise`，`promise` 解决之后会返回一个包含所有在过去 24 小时里登录过的用户的集合。

9. 扩展子类

`Bookshelf` 的 `extend()` 方法正确地设置了原型链。因此，除了可以创建直接继承自 `Model` 类的模型之外，开发者也可以创建继承自别的模型的模型，如清单 12-36 所示。

清单 12-36 创建一个直接继承自 `Model` 类的 `Base` 类，从而其他模型也可以继承

```
// example-bookshelf-extend/lib/base.js
/**
 * 该模型是一个基类，应用中所有其他的模型都可以继承
 *
 * @class Base
 */
var Base = bookshelf.Model.extend({
  'initialize': function() {
```

```

    this._initEventBroadcasts();
  },
  'foo': function() {
    console.log('bar', this.toJSON());
  }
});

```

```
// example-bookshelf-extend/lib/user.js
```

```

/**
 * @class User
 */
var User = Base.extend({
  'tableName': 'users'
});

```

```
// example-bookshelf-extend/index.js
```

```

var User = require('./lib/user');
User.where({
  'id': 1
}).fetch().then(function(user) {
  user.foo();
});

```

可以创建出多级继承的模型给我们提供了一些实用的机遇。大多数实用 Bookshelf 的应用遵循清单 12-36 中的方式，也就是先创建一个基类，然后应用中所有其他的模型都继承自这个基类。按照这种模式，我们可以很轻易地通过修改基类来增加核心功能。清单 12-36 中，User 模型（以及其他所有继承自 Base 的模型）会继承 Base 模型中的 foo() 方法。

12.2.3 关系

像 Bookshelf 这样的 ORM 库提供了简便的面向对象的模式，让我们和存储于扁平的关系型数据库表中的数据进行交互。通过 Bookshelf 的帮助，我们可以指定不同的应用模型之间的关系。例如，一个账户可能有多个用户，或者一个用户可能有多个书签。一旦这些关系定义了之后，我们就能使用 Bookshelf 的新方法。这些方法能让我们更方便地理清这些关系。

表 12-1 列出了一些常用的关系。

表 12-1

常用的 Bookshelf 关系

关联关系	关系类型	例子
一对一	hasOne	一个用户有一个个人主页
一对一	belongsTo	一个个人主页属于一个用户
一对多	hasMany	一个账户有很多个用户
一对多	belongsToMany	一个用户属于一个账户
多对多	belongsToMany	一本书有一个或多个作者

在接下来的部分中，你会发现这些关系之间的区别、它们是如何定义的，以及如何在一个应用中很好地使用它们。

1. 一对一

一对一关联是最简单的一种形式。正如它的名字，一对一关联表示一个特定的模型和另一个模

型关联。根据我们叙述模型相对顺序的不同，这种关联可以是 `hasOne` 关系，也可以是 `belongsTo` 关系。

图 12-3 展示了例子背后的数据库模式。在这个例子中，`profile` 表有一个 `user_id` 外键列，它与 `users` 表相关联。

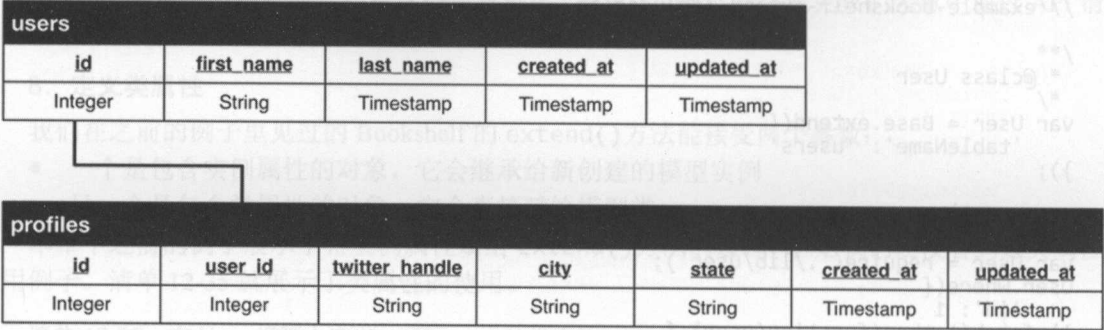


图 12-3 一对一关系下的数据库模式

hasOne 和 belongsTo

`hasOne` 关系表示一个模型“含有”另一个模型；而 `belongsTo` 则相反，表示一个模型“属于”另一个模型。换句话说，`belongsTo` 和 `hasOne` 是相反的关系。清单 12-37 展示了用 Bookshelf 定义这些关系的过程。

清单 12-37 用 Bookshelf 定义 hasOne 和 belongsTo 关系
// example-bookshelf-relationships1/lib/user.js

```
/**
 * @class User
 *
 * 一个用户有一个个人主页
 */
var User = bookshelf.Model.extend({
  'tableName': 'users',
  /**
   * Bookshelf relationships are defined as model instance
   * methods. Here, we create a 'profile' method that will
   * allow us to access this user's profile. This method
   * could have been named anything, but in this case -
   * 'profile' makes the most sense.
   */
  /**
   * Bookshelf 的关系是通过模型实例方法定义的。在这里，我们创建了
   * 一个“profile”方法，它可以让我们访问这个用户的个人主页。这个方
   * 法可以起任何名字，但在这里，“profile”这个名字最合适。
   */
  'profile': function() {
    return this.hasOne(Profile);
  }
});

// example-bookshelf-relationships1/lib/profile.js

/**
 * @class Profile
```



```

*
* 一个个人主页属于一个用户
*/
var Profile = bookshelf.Model.extend({
  'tableName': 'profiles',
  'user': function() {
    return this.belongsTo(User);
  }
});

```

Bookshelf 是通过使用特定的实例函数来定义关系的，如清单 12-37 所示。这些关系定义好之后，就可以简便地使用它们了。对于初学者，清单 12-38 展示了加载一个模型已经初始化过的关系的过程。例子运行的输出结果如清单 12-39 所示。

清单 12-38 加载一个模型已经初始化过的关系

```

// example-bookshelf-relationships1/index.js

User.where({
  'id': 1
}).fetch().then(function(user) {
  return user.load('profile');
}).then(function(user) {
  console.log(JSON.stringify(user.toJSON(), null, 4));
});

```

清单 12-39 清单 12-38 的输出结果

```

{
  "id": 1,
  "first_name": "Steve",
  "last_name": "Taylor",
  "created_at": "2014-10-02",
  "profile": {
    "id": 1,
    "user_id": 1,
    "twitter_handle": "staylor",
    "city": "Portland",
    "state": "OR",
    "created_at": "2014-10-02"
  }
}

```

在清单 12-38 中，我们首先获取 User 模型的实例。之后，模型默认的行为是返回自身的信息，而不是和它相关联的模型的信息。因此，在这个例子中，我们必须首先通过 load() 方法加载关联模型 Profile，方法返回的是一个 promise，当模型被获取之后就会进入它的解决分支。然后，我们就可以使用用户相关的实例方法来引用这个用户的个人主页信息。

如果我们使用渴望加载 (eager loading)，如清单 12-40 所示，Bookshelf 的关系会变得更加有用。在这个例子中，我们获取 User 模型的同时也获取了 Profile 模型。我们给 fetch() 方法传一个选项对象，对象里指定一个或多个我们感兴趣的关系。返回的 promise 进入解决分支之后，会得到一个已经包含 profile 关系的 User 模型实例。

清单 12-40 利用“渴望加载”，通过仅仅一个调用，就能获取我们的 User 模型以及和它关联的 Profile 模型

```

// example-bookshelf-relationships1/eager.js

```

```

User.where({
  'id': 1

```

```
}).fetch({
  'withRelated': ['profile']
}).then(function(user) {
  console.log(JSON.stringify(user.toJSON(), null, 4));
});
```

2. 一对多

一对多关联构成了大多数常见的关系的基础。这种关联基于简单的一对一关联之上，并且能让我们将一个模型和多个其他的模型关联。它有 `hasMany` 和 `belongsTo` 两种关系形式，接下来我们就能看到。

图 12-4 展示了我们即将要研究的例子背后的数据库模式。在这个例子中，`user` 表含有一个 `account_id` 外键列，它和 `accounts` 表关联。

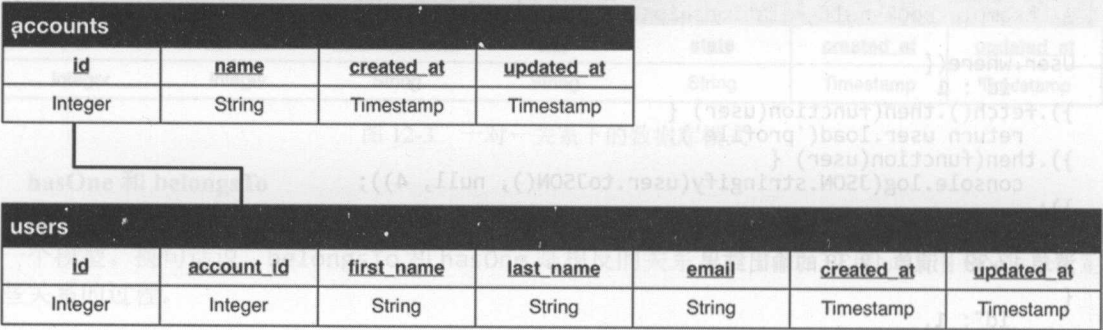


图 12-4 一对多关系下的数据库模式

`hasMany` 和 `belongsTo`

`hasMany` 关系表示一个模型可能含有多个（或者一个也没有）同种类型的其他模型。在之前的例子里出现过的 `belongsTo` 关系，对一对多关联的情况也适用。清单 12-41 展示了用 Bookshelf 定义这些关系的过程。清单 12-42 展示了它们的用法。

清单 12-41 定义 `hasMany` 和 `belongsTo` 关系

```
// example-bookshelf-relationships2/lib/account.js
/**
 * @class Account
 *
 * 一个账户含有一个或多个用户
 */
var Account = bookshelf.Model.extend({
  'tableName': 'accounts',
  'users': function() {
    return this.hasMany(User);
  }
});

// example-bookshelf-relationships2/lib/user.js
/**
 * @class User
 *
 * 一个用户属于一个账户
 * 一个用户有一个个人主页
```

```

*/
User = bookshelf.Model.extend({
  'tableName': 'users',
  'account': function() {
    return this.belongsTo(Account);
  },
  'profile': function() {
    return this.hasOne(Profile);
  }
});

// example-bookshelf-relationships2/lib/profile.js

```

```

/**
 * @class Profile
 *
 * 一个个人主页属于一个用户
 */
Profile = bookshelf.Model.extend({
  'tableName': 'profiles',
  'user': function() {
    return this.belongsTo(User);
  }
});

```

清单 12-42 加载账户模型的实例，以及所有和它相关联的用户

```

// example-bookshelf-relationships2/index.js

Account.where({
  'id': 1
}).fetch({
  'withRelated': ['users']
}).then(function(account) {
  console.log(JSON.stringify(account.toJSON(), null, 4));
});

{
  "id": 1,
  "name": "Acme Company",
  "created_at": "2014-10-02",
  "users": [
    {
      "id": 1,
      "account_id": 1,
      "first_name": "Steve",
      "last_name": "Taylor",
      "email": "steve.taylor@mydomain.com",
      "created_at": "2014-10-02"
    },
    {
      "id": 2,
      "account_id": 1,
      "first_name": "Sally",
      "last_name": "Smith",
      "email": "sally.smith@mydomain.com",
      "created_at": "2014-10-02"
    }
  ]
}

```

在清单 12-42 中，我们看到了 Bookshelf 的“渴望加载”的另一个例子。利用它，我们可以在获

取模型的同时获取和它关联的所有模型。当你发现还可以用“渴望加载”来加载嵌套关系（那些存在于对象之中的更深层的关系）时，你就会觉得这个“渴望加载”的概念变得越来越有意思了。只有当我们开始使用 Bookshelf 的“渴望加载”特性时，我们才能体会到 Bookshelf 以及其他类似的 ORM 工具所提供的“虚拟对象数据库”的好处。清单 12-43 中的例子可以帮助我们更好地理解这个概念。

清单 12-43 通过“渴望加载”的方法加载一个账户以及它所有的用户和每个用户对应的个人主页
// example-bookshelf-relationships2/nested-eager.js

```
Account.where({
  'id': 1
}).fetch({
  'withRelated': ['users', 'users.profile']
}).then(function(account) {
  console.log(JSON.stringify(account.toJSON(), null, 4));
});
```

```
/*
{
  "id": 1,
  "name": "Acme Company",
  "created_at": "2014-10-02",
  "users": [
    {
      "id": 1,
      "account_id": 1,
      "first_name": "John",
      "last_name": "Doe",
      "email": "john.doe@domain.site",
      "created_at": "2014-10-02",
      "profile": {
        "id": 1,
        "user_id": 1,
        "twitter_handle": "john.doe",
        "city": "Portland",
        "state": "OR",
        "created_at": "2014-10-02"
      }
    },
    {
      "id": 2,
      "account_id": 1,
      "first_name": "Sarah",
      "last_name": "Smith",
      "email": "sarah.smith@domain.site",
      "created_at": "2014-10-02",
      "profile": {
        "id": 2,
        "user_id": 2,
        "twitter_handle": "sarah.smith",
        "city": "Asheville",
        "state": "NC",
        "created_at": "2014-10-02"
      }
    }
  ]
}
*/
```


3. 多对多

多对多关联和一对一关联、一对多关联的不同之处在于，它可以让一条记录和一条或多条不同类型的记录相关联。为了理清这个概念，请看图 12-5，图中阐释了一个常用的关于作者和书的例子。

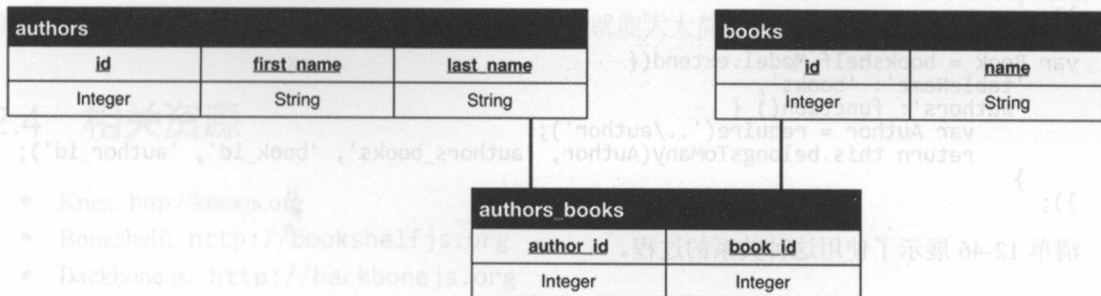


图 12-5 多对多关系使用了第三个连接表。在这个例子中，一个作者可以写多本书，一本书可以有多个作者

前面的例子中的单个外键列（见图 12-5）已经不够用了。为了表现多对多的关系，我们需要引入第三个连接表（authors_books）。

belongsToMany

图 12-5 中的数据库模式可以通过 Bookshelf 的 belongsToMany 关系描述，如清单 12-44 所示。

清单 12-44 用 Bookshelf 表示 belongsToMany 关系

```
// example-bookshelf-relationships3/lib/author.js
```

```
var Author = bookshelf.Model.extend({
  'tableName': 'authors',
  'books': function() {
    return this.belongsToMany(require('./book'));
  }
});
```

```
// example-bookshelf-relationships3/lib/book.js
```

```
var Book = bookshelf.Model.extend({
  'tableName': 'books',
  'authors': function() {
    return this.belongsToMany(require('./author'));
  }
});
```

需要注意的是，使用 belongsToMany 关系时，Bookshelf 会自动根据你的数据库模式做一些假设——除非你告诉 Bookshelf 不要这么做。Bookshelf 会假设：

- 第三个连接表是存在的，它的表名由两个相关联的表派生出来，用下划线分隔，并且表名按字母表顺序排序，在这个例子中是 authors_books。
- 连接表中的列名是由两个相关联的表名的单数形式加上_id 后缀派生出来的，在这个例子中是 author_id 和 book_id。

如果你不想使用默认的命名规则，可以通过更改调用 this.belongsToMany 的方式实现，如清单 12-45 所示。

清单 12-45 用 Bookshelf 描述 belongsToMany 关系，同时提供特定的表名和列名

```
var Author = bookshelf.Model.extend({
  'tableName': 'authors',
  'books': function() {
    return this.belongsToMany(
      require('./book'), 'authors_books', 'author_id', 'book_id');
  }
});

var Book = bookshelf.Model.extend({
  'tableName': 'books',
  'authors': function() {
    var Author = require('./author');
    return this.belongsToMany(Author, 'authors_books', 'book_id', 'author_id');
  }
});
```

清单 12-46 展示了使用这种关系的过程。

清单 12-46 清单 12-45 中定义的关系的使用方法以及输出结果

// example-bookshelf-relationships3/index.js

```
Book.fetchAll({
  'withRelated': ['authors']
}).then(function(books) {
  console.log(JSON.stringify(books.toJSON(), null, 4));
});
```

```
/*
[
  {
    id: 1,
    name: 'Pro JavaScript Frameworks for Modern Web Development',
    authors: [{
      id: 1,
      first_name: 'Tim',
      last_name: 'Ambler',
      _pivot_book_id: 1,
      _pivot_author_id: 1
    }, {
      id: 2,
      first_name: 'Nicholas',
      last_name: 'Cloud',
      _pivot_book_id: 1,
      _pivot_author_id: 2
    }
  ]
}]
*/
```

12.3 小结

如果你调研过过去几年的数据库市场的格局，你会很容易发现所谓的“NoSQL”存储平台已经大面积地代替了像 MySQL 和 PostgreSQL 这样的老旧的关系型数据库，但是这并不能说明更多东西。正如马克·吐温在 1897 年被过早地误报逝世一样，关系型数据库的灭亡也是一种夸大。

关系型数据库提供了很多很炫的特性，可惜大部分并不在本章的讨论范围里。很多精彩的书籍已经写过这个观点，我建议大家了解这些特性，然后决定如何存储项目数据以及存储在何种数据库

平台。其中一个重要的特性（本章中有提及）是对事务的支持，也就是多个查询可以被组合成单个工作单元，它们要么全部成功，要么全部失败。清单 12-13 和清单 12-14 中的金融交易的例子展示了事务这个概念在一些关键任务的应用中所扮演的重要角色。

Knex 提供的跨平台的 API，以及它基于 promise、支持事务、提供迁移管理的特性，给开发者提供了一个简便的工具和关系型数据库进行交互。当它和 Bookshelf（一个对熟悉 Backbone 的开发者来说能很快熟悉的 ORM）配合时，处理复杂数据的过程就能大大简化。

12.4 相关资源

- Knex: <http://knexjs.org>
- Bookshelf: <http://bookshelfjs.org>
- Backbone.js: <http://backbonejs.org>
- Underscore.js: <http://underscorejs.org>
- MySQL: www.mysql.com
- PostgreSQL: www.postgresql.com
- MariaDB: <http://mariadb.org>
- SQLite: www.sqlite.org
- Multiline: <https://github.com/sindresorhus/multiline>

Faye

因特网上的名人名言有个问题，就是你永远没办法确定它们是不是真实的。

——亚伯拉罕·林肯

基于网络的应用在最近几年里变得越来越复杂，而这主要是源于对现代网络开发技术——如 HTML、WebSockets 以及新的 JavaScript 标准 API（如 Geolocation、Web Storage 和 Web Audio）的大规模使用。曾经那些只能在传统的桌面应用领域使用的功能现在可以用到浏览器上了，这让网络开发者们能开发出短短几年前还完全不可能实现的应用。

但是，随着网络浏览器的日渐成熟，传统的网络和 HTTP 协议开始显得越来越老旧。那种简单的“请求-响应”的通信模式，即客户端（如网页浏览器）发送请求给服务器端获取资源，然后服务器端响应请求的模式，已经不再能很好地处理今天大多数创新性网页应用所面临的实时性问题了。需要快速地处理不断变化的应用（如多人游戏、社交网站以及聊天室），对“数据推送”有着强烈的需求。所谓“数据推送”，就是指从服务端向客户端发起通信。

网页浏览器日渐成熟的同时，使用它们的用户的期望也变得越来越成熟。那些只能在用户发起请求时才会给用户所需求的消息的网络应用，很快就会被那些在用户感兴趣的事情发生时能主动提醒用户的应用所替代。

本章中会讲解 Faye。它可以同时在 Node.js 和浏览器中使用，给开发者提供一个开发实时通信应用时会用到的强大的工具集。具体的主题有

- HTTP、Bayeux 和 WebSocket
- 使用 Faye 的发布/订阅通道来通信
- 开发 Faye 扩展
- 安全管理

13.1 HTTP、Bayeux 和 WebSocket

HTTP 协议被称作“请求-响应”协议，是因为客户端向服务端请求资源后会接收到服务端的响应。HTTP 协议是“无状态”的，因为每一个“请求-响应”信息对都是独立的，不同请求不会保留各自的“状态”（记忆）。图 13-1 阐释了这个概念。

HTTP 的这种符合直觉的通信模式使得互联网在今天能获得如此广泛的应用。但是协议的简洁性也带来巨大的成本，它在处理双向、异步的事件驱动消息时显得力不从心。



图 13-1 HTTP 协议是无状态的请求-响应协议

举个例子，想象有多个用户在一个聊天室里参与聊天的场景（见图 13-2）。当每个用户发送新消息时，其他用户需要立即被告知，但是 HTTP 的请求-响应的特性没办法让这些消息事件从服务端主动发送到客户端，如图 13-3 所示。

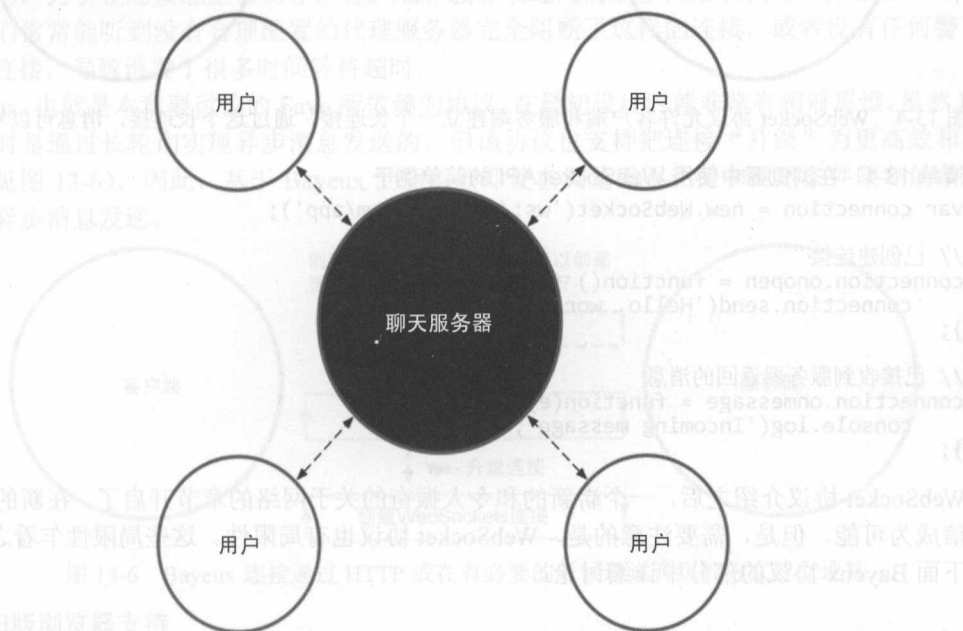


图 13-2 一个时间驱动的消息平台，当别的同伴发送消息时，用户立即就能收到消息

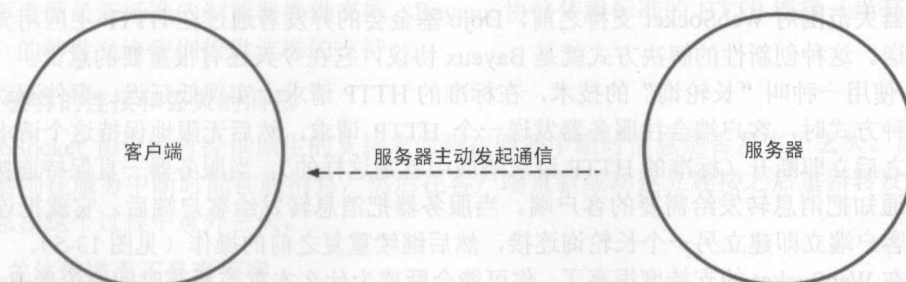


图 13-3 HTTP 协议不允许服务端主动向客户端发起通信

13.1.1 WebSocket

和 HTTP 协议不同，WebSocket 协议是专门针对在浏览器和远程服务器之间建立一个全双工（双向）的 TCP 长连接而设计的（见图 13-4）。该协议在 2011 年成为标准（RFC 6455），目前在大多数流行的浏览器的近期版本上都支持。因此，支持的浏览器现在就可以真正地采用异步的方式和服务端通信了（假设服务器支持这样的连接）。清单 13-1 展示了一个使用 WebSocket API 的简单例子。

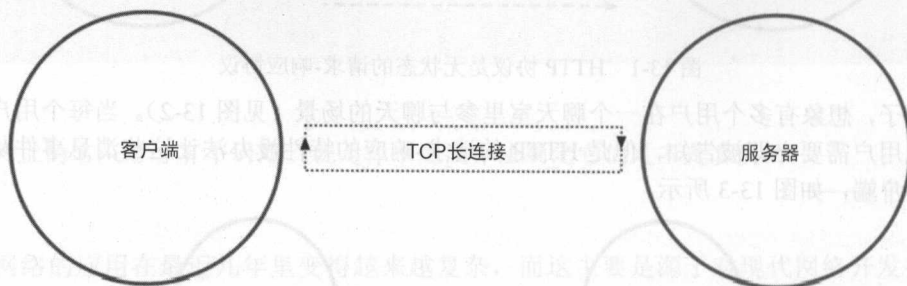


图 13-4 WebSocket 协议允许客户端和服务端建立一个长连接，通过这个长连接，消息可以双向传递

清单 13-1 在浏览器中使用 WebSocket API 的简单例子

```
var connection = new WebSocket('ws://domain.com/app');

// 已创建连接
connection.onopen = function() {
    connection.send('Hello, world.');
```

```
};

// 已接收到服务器返回的消息
connection.onmessage = function(e) {
    console.log('Incoming message', e);
};
```

WebSocket 协议介绍之后，一个崭新的和令人振奋的关于网络的章节开启了。在新的章节中，实时通信成为可能。但是，需要注意的是，WebSocket 协议也有局限性。这些局限性乍看之下不明显，将在下面 Bayeux 协议的部分中详细讨论。

13.1.2 Bayeux 协议

在浏览器大范围对 WebSocket 支持之前，Dojo 基金会的开发者通过在 HTTP 上应用异步事件驱动的消息发送。这种创新性的解决方式就是 Bayeux 协议，它在今天还有很重要的意义。

Bayeux 使用一种叫“长轮询”的技术，在标准的 HTTP 请求上实现低延迟、事件驱动的消息发送。使用这种方式时，客户端会往服务器发送一个 HTTP 请求，然后无限地保持这个请求，而不是在响应返回之后立即断开（标准的 HTTP 请求响应中就是这样的）。当服务器一直保持连接时，它其实在等待被通知把消息转发给需要的客户端。当服务器把消息转发给客户端后，它就把连接关闭。在这之后，客户端立即建立另一个长轮询连接，然后继续重复之前的操作（见图 13-5）。

既然现在 WebSocket 的支持度很高了，你可能会疑惑为什么本章节要花时间讨论像 Bayeux 这样看似过时的概念。然而，Bayeux 并不完全像你认为的那样过时。虽然 WebSocket 很神奇，但是它们

有局限性。在这种局限的情况下，Bayeux 就成为替代方案来解决问题。

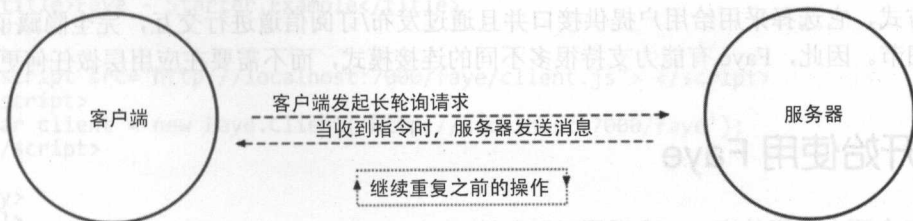


图 13-5 通过 Bayeux 的长轮询，基于 HTTP 的异步消息发送成为可能

1. 网络挑战

尽管大多数流行的浏览器的近期版本现在已经支持 WebSocket，但是还是不能保证所有的客户端和服务端之间的连接能成功建立。WebSocket 协议设计的长连接和网络最初就有的“请求-响应”的短连接不同。为了让连接能正常工作，客户端和服务端之间的众多网络 and 代理服务器都必须合理地配置。我们常常能听到没有合理配置的代理服务器完全阻断了这样的连接，或者没有任何警告就丢弃了这些连接，导致浪费了很多时间等待超时。

Bayeux，也就是本章要讨论的 Faye 所依赖的协议，在最初设计时就非常有超前思维。虽然 Bayeux 最初设计时是通过长轮询实现异步消息发送的，但该协议也支持把连接“升级”为更高效和更现代的标准（见图 13-6）。因此，基于 Bayeux 的服务器即使在浏览器不支持和网络条件限制的条件下，也能实现异步消息发送。

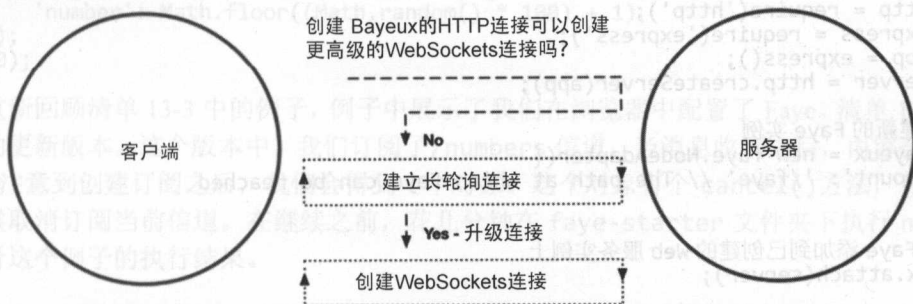


图 13-6 Bayeux 连接通过 HTTP 或在有必要的时候通过升级的方式建立连接

2. 旧版浏览器支持

之前说过，所有主流浏览器的最新版本都支持 WebSocket 协议。根据应用的需要，你可能需要为那些不遵循最新标准的浏览器提供支持。Bayeux 协议依赖标准的 HTTP 连接（当有需要的时候可以升级）的特性允许我们提供这样的支持。

3. 丢弃的连接和丢失的消息

WebSocket 协议没有提供原生的支持来侦测在服务中断时信息是否发生了丢失。而基于 Bayeux 的服务器则在服务中断时能挂起消息，然后在客户端重新成功建立连接之后重新转发消息，避免重要的信息在这个过程中意外丢失。

4. 关注信道而不是套接字

WebSocket 的 API 给开发者提供了简单和清晰定义的接口，在客户端和服务端之间建立套接字

连接和在它们之间传送消息（下面将讨论）。但是它不提供任何高级的抽象来管理交互。Faye 采用了另外一种方式，它选择采用给用户接口并且通过发布/订阅信道进行交互，完全隐藏了内部网络层的工作细节。因此，Faye 有能力支持很多不同的连接模式，而不需要在应用层做任何更改。

13.2 开始使用 Faye

Faye 包含两个分开的库：一个适用于服务端（Node.js），另一个适用于浏览器端。在继续之前，我们先看一下在服务端安装和配置 Faye 的基本步骤。

下面的例子展示了通过 npm 命令行在一个 Node.js 项目中安装 Faye：

```
$ npm install faye --save
```

Faye 和 Node 的原生 http 和 https 模块一起配合工作，创建出一个端口来接收连接。这个过程展示在了清单 13-2 中。在这个例子中，我们创建一个网络服务来伺服项目中的 public 文件夹下的静态文件。然后 Faye 的一个实例被创建并赋给 Node.js 服务，从而能让客户端通过同一个端口连接到网络服务和 Faye 上。

清单 13-2 在 Node.js 中初始化 Faye

```
// faye-starter/lib/server.js
var faye = require('faye');
var http = require('http');
var express = require('express');
var app = express();
var server = http.createServer(app);

// 创建新的 Faye 实例
var bayeux = new faye.NodeAdapter({
  'mount': '/faye' // The path at which Faye can be reached
});

// 将 Faye 添加到已创建的 Web 服务实例上
bayeux.attach(server);

// 在项目目录 '/public' 输出静态文件
app.use('/', express.static(__dirname + '/../public'));

// Web 服务和 Faye 均可通过 7000 端口监听
server.listen(7000);
```

现在我们的服务器能够接收连接了，接下来开始在浏览器里配置 Faye（见清单 13-3）。我们首先创建一个 script 标签，它会加载 Faye 的客户端库（client.js），该文件可以在清单 13-2 中的挂载路径访问到。接下来，我们创建一个新的客户端并配置该客户端，让它连接到那个挂载路径。

清单 13-3 在浏览器中配置 Faye

```
// faye-starter/public/index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```



```

<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Faye - Starter Example</title>
</head>
<body>
  <script src="http://localhost:7000/faye/client.js"> </script>
  <script>
    var client = new Faye.Client('http://localhost:7000/faye');
  </script>

</body>
</html>

```

13.3 发布/订阅消息系统

与其给开发者提供 API 直接对套接字连接进行交互, Faye 更推荐使用发布/订阅主题信道。为了说明这个过程, 重新回顾清单 13-2 (这个例子中, 我们配置了服务), 然后加上清单 13-4 中的代码。加上这段代码之后, 让我们的服务每两秒钟发布一个随机的数字到 `/number` 信道。

清单 13-4 每两秒钟发布一条消息到 `/number` 信道

```

// faye-starter/lib/server.js

setInterval(function() {
  // Pass a topic channel, along with a payload of data
  bayeux.getClient().publish('/numbers', {
    'number': Math.floor((Math.random() * 100) + 1)
  });
}, 2000);

```

现在重新回顾清单 13-3 中的例子, 例子中展示了我们在浏览器中配置了 Faye。清单 13-5 展示了这个例子的更新版本。这个版本中, 我们订阅了 `/numbers` 信道。当消息收到之后, 内容会被添加到 DOM 中。注意到创建订阅之后, 我们会得到一个对象。这个对象有个 `cancel()` 方法, 可以让我们在任何时候取消订阅当前信道。在继续之前, 花几分钟在 `faye-starter` 文件夹下执行 `npm start` 命令来看看这个例子的执行结果。

清单 13-5 在浏览器中 `/numbers` 信道上侦听消息

```

// faye-starter/public/index.html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Faye - Starter Example</title>
</head>
<body>
  <a href="#" id="cancelbt">Cancel Subscription to `/numbers` channel</a>

  <ul id="container"></ul>

  <script src="http://localhost:7000/faye/client.js"></script>
  <script src="/bower_components/jquery/dist/jquery.js"></script>

  <script>

```

```

var client = new Faye.Client('http://localhost:7000/faye');

var subscription = client.subscribe('/numbers', function(data) {
  console.log('Incoming message on the `numbers` channel', data);
  $('#container').append('<li>' + data.number + '</li>');
});

$('#cancelbt').one('click', function() {
  console.log('Canceling subscription to `numbers` channel');
  subscription.cancel();
});
</script>

</body>
</html>

```

在清单 13-4 中，我们可以看到消息如何在服务端发布消息。客户端发布自己的消息的过程和服务端的几乎一样。清单 13-6 中的例子展示了在浏览器客户端中向 /foo 信道发送 Hello, world 消息。

清单 13-6 从浏览器发布消息

```

client.publish('/foo', {
  'text': 'Hello, world.'
}).then(function() {
  // Message was received by server
});

```

信道通配符

除了能在特定的信道上订阅消息，Faye 客户端也可以通过给 `subscribe()` 方法传一个通配符来订阅多个信道，如清单 13-7 所示。

清单 13-7 使用通配符订阅符合特定模式的多个信道

```

/**
 * 订阅单个信道段的消息。任何来自 '/foo' 直接下一级信道的消息都会给记录
 */
client.subscribe('/foo/*', function(message) {
  console.log('Message received', message);
});

/**
 * 订阅所有来自 '/foo' 下面的信道段的消息。
 */
client.subscribe('/foo/**', function(message) {
  console.log('Message received', message);
});

```

Faye 对信道通配符的支持可以使很多事成为可能，包括给某个特定的用户（或一组用户）创建特定命名空间化主题的信道。例如，假想有一个应用以图 13-7 所示的方式组织用户。

在这个应用中，每个用户属于一个父级账户（可以含有多个用户）。有了这样的模式之后，我们很容易想象可能会有想给某个特定用户或某个特定账户下的所有用户发送消息。通过通配符订阅和命名空间化的信道，我们可以轻松地完成这个目标。

本章中的 `faye-security` 项目（见图 13-8）基于之前讲述过的若干个主题（AngularJS, Knex 和 Bookshelf）构建了一个应用。该应用可以让注册用户登录并测试各种 Faye 的功能，包括通配符订

阅和命名空间化的信道。

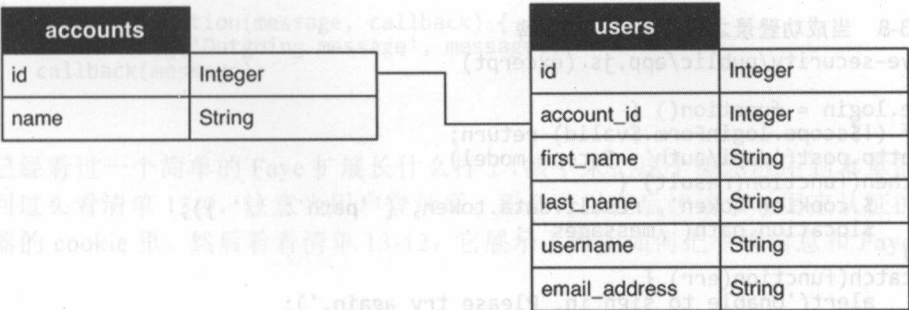


图 13-7 一个有多个用户同属于某父级账户下的应用

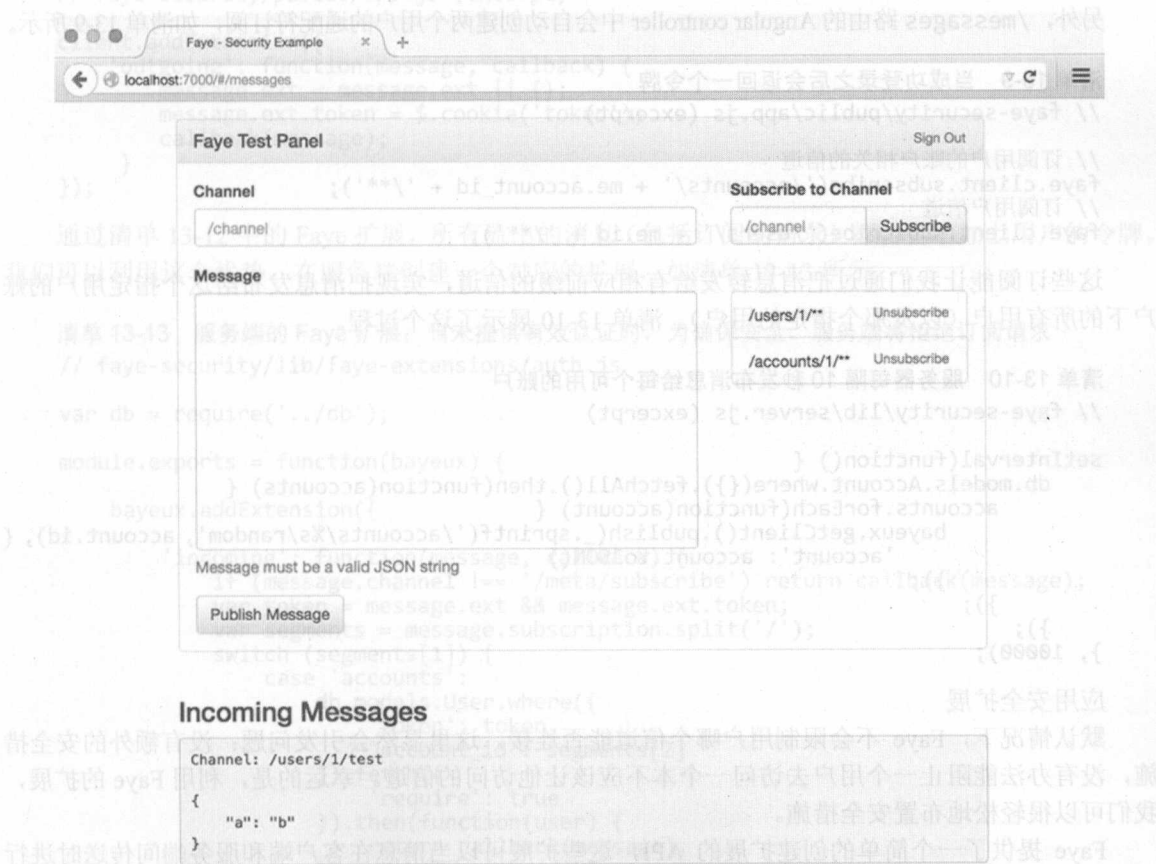


图 13-8 登录之后，用户可以在界面中管理信道订阅和发布消息

当登录表单提交之后，浏览器会发送一个请求到服务端来验证用户输入的用户名和密码。如果请求成功，服务器会返回一个对象，对象中会包含认证过的用户的信息，包括会给一个令牌，后续的请求都会使用这个令牌来认证。清单 13-8 展示了/login 路由的 Angular controller 的代码片段（当

请求该路由时将执行这段 controller)。

清单 13-8 当成功登录之后会返回一个令牌

```
// faye-security/public/app.js (excerpt)

$scope.login = function() {
  if (!$scope.loginForm.$valid) return;
  $http.post('/api/auth', $scope.model)
    .then(function(result) {
      $.cookie('token', result.data.token, { 'path': '/' });
      $location.path('/messages');
    })
    .catch(function(err) {
      alert('Unable to sign in. Please try again.');
```

另外，/messages 路由的 Angular controller 中会自动创建两个用户的通配符订阅，如清单 13-9 所示。

清单 13-9 当成功登录之后会返回一个令牌

```
// faye-security/public/app.js (excerpt)

// 订阅用户的账户相关的信道
faye.client.subscribe('/accounts/' + me.account_id + '/*');
// 订阅用户信道
faye.client.subscribe('/users/' + me.id + '/*');
```

这些订阅能让我们通过把消息转发给有相应前缀的信道，实现把消息发布给这个指定用户的账户下的所有用户（以及那个指定的用户）。清单 13-10 展示了这个过程。

清单 13-10 服务器每隔 10 秒发布消息给每个可用的账户

```
// faye-security/lib/server.js (excerpt)

setInterval(function() {
  db.models.Account.where({}).fetchAll().then(function(accounts) {
    accounts.forEach(function(account) {
      bayeux.getClient().publish(_sprintf('/accounts/%s/random', account.id), {
        'account': account.toJSON()
      });
    });
  });
}, 10000);
```

应用安全扩展

默认情况下，Faye 不会限制用户哪个信道能否连接。这里显然会引发问题：没有额外的安全措施，没有办法能阻止一个用户去访问一个本不应该让他访问的信道。幸运的是，利用 Faye 的扩展，我们可以很轻松地布置安全措施。

Faye 提供了一个简单的创建扩展的 API，这些扩展可以当消息在客户端和服务端间传送时进行拦截分析和（视情况而定）修改消息。这样的扩展既可以在服务端创建，也可以在浏览器中创建。过程是一样的，不管它们是在哪儿创建的。清单 13-11 展示了 Faye 扩展的一个简单的例子。

清单 13-11 通过简单的 Faye 扩展将所有进入的和流出的消息记录到控制台

```
client.addExtension({
  'incoming': function(message, callback) {
```



```

        console.log('Incoming message', message);
        callback(message);
    },
    'outgoing': function(message, callback) {
        console.log('Outgoing message', message);
        callback(message);
    }
});

```

我们已经看过一个简单的 Faye 扩展长什么样了,接下来把这个概念应用到本章的样例 Web 应用里。回过头看清单 13-9,注意当用户登录后,那个可以在后续请求时用来认证的令牌被存储在浏览器的 cookie 里。然后看看清单 13-12,它展示了应用如何把令牌信息和 Faye 扩展结合起来。

清单 13-12 一个能将用户的令牌自动附到所有流出消息的扩展

// faye-security/public/app.js (excerpt)

```

client.addExtension({
  'outgoing': function(message, callback) {
    message.ext = message.ext || {};
    message.ext.token = $.cookie('token');
    callback(message);
  }
});

```

通过清单 13-12 中的 Faye 扩展,所有流出的消息(包括订阅的请求)都会自动加上用户的令牌。我们可以利用这个优势,在服务端创建一个对应的扩展,如清单 13-13 所示。

清单 13-13 服务端的 Faye 扩展:当未提供有效认证时,为确保安全,服务端将拒绝订阅请求

// faye-security/lib/faye-extensions/auth.js

```

var db = require('../db');

module.exports = function(bayeux) {
  bayeux.addExtension({
    'incoming': function(message, callback) {
      if (message.channel !== '/meta/subscribe') return callback(message);
      var token = message.ext && message.ext.token;
      var segments = message.subscription.split('/');
      switch (segments[1]) {
        case 'accounts':
          db.models.User.where({
            'token': token,
            'account_id': segments[2]
          }).fetch({
            'require': true
          }).then(function(user) {
            return callback(message);
          }).catch(function(err) {
            message.error = 'Permission denied.';
            return callback(message);
          });
          break;
        case 'users':
          db.models.User.where({
            'token': token,

```

```

        'id': segments[2]
    }).fetch({
        'require': true
    }).then(function(user) {
        return callback(message);
    }).catch(function() {
        message.error = 'Permission denied.';
        return callback(message);
    });
    break;
default:
    return callback(message);
break
}
});
};

```

当服务端在 `/meta/subscribe` 信道收到消息后，清单 13-13 中的扩展会校验想要订阅的信道是不是属于安全的命名空间下（如 `/accounts` 和 `/users`）。如果是的话，扩展会校验是否有令牌，然后用该令牌的信息去数据库中查找对应的用户。如果找到对应的用户，消息就能继续进行后续操作；否则，订阅过程就会被拒绝，错误信息会附在消息对象的 `error` 属性中。

13.4 小结

在这一章中，你熟悉了很多概念，这些概念能让你创建近乎实时的基于浏览器的通信应用。你成功在服务端和浏览器里安装和配置了 Faye。你知道了 Faye 是如何使用订阅/发布信道在客户端之间传输消息的。然后你进一步使用了命名空间化的信道和使用通配符订阅信道。你还学习了如何扩展 Faye，从而在浏览器和服务端传输消息时能监控、修改和（视情况而定）拒绝消息。

根据我们的经验，在网络上创建异步、事件驱动的通信应用的最佳实践包括利用 WebSocket 的优势、拥有备用解决方案以及提供上层抽象层避免让开发者考虑网络层的东西。Faye 把 WebSocket 的速度和效率与 Bayeux 协议的稳定性整合起来，构建了一套健壮的方式来实现异步、事件驱动通信，很好地满足了最佳实践的要求。

13.5 相关资源

Faye: <http://faye.jcoglan.com/>

The Dojo Foundation: <http://dojofoundation.org/>

The Bayeux Protocol: <http://svn.cometd.org/trunk/bayeux/bayeux.html>

Q

我是一个创意人。努力工作不是我的强项。

——Q,《星际旅行：航海家号》

JavaScript 是一门异步编程语言。无论是在浏览器端还是服务器端，开发者都可以控制 JavaScript 的运行，“安排”代码在未来的某个时刻执行。这个特性常用来延缓 CPU 密集型或长时运行的操作的执行，让程序在开始执行下一个任务前有足够的完成当前任务。这个特性很强大，很多传统的同步编程语言如 Java、C#、PHP 以及 Ruby 都跟风并采纳。一些语言已经实现了异步执行模型并将其作为语言的一个特性，如 C#（通过 `async` 和 `await` 关键字）；其他语言通过依赖外部库来支持异步特性，如 PHP 的 React 库（不要和 Facebook 的 JavaScript 库 React 混淆）。无论在哪种情况下，同步代码和异步代码一定会同时出现。

Q 是一个 JavaScript 库，它将异步行为封装在一个接口中，使得代码读起来很像同步代码，这是本章的主题。Q 提供了一种专门的对象 `Promise`，可级联起来消除嵌套回调，并很好地处理异步代码中的值传递、错误处理及控制流等问题。但在深入研究 Q 之前，我们有必要走一段弯路来证明编写和维护异步代码是一件困难的事。

14.1 时间就是一切

同步代码读起来简单，是因为计算机一次只执行一条语句，产生的（如方法执行）返回值可以立即被使用。有模式化异常处理特性的语言提供了 `try/catch/finally` 块，可用来预防错误，并在出错的时候处理，以免细微的错误酿成程序的致命错误。但是模式化异常处理特性只在同步代码中起作用。它的行为就像 `goto` 语句，使代码“跳”到程序的其他地方，并从该处恢复语句的执行。

而异步代码的行为就有点不一样了。在 JavaScript 中，异步代码被安排在未来的某个时刻执行（有时候刚好在当前执行代码执行后）。这不同于同步模型，因为后面的代码执行只能在前堆栈展开之后。同时，异步代码所产生的返回值和错误异常也必须在其实际执行的时候才能被处理。

许多语言（包括 JavaScript）使用回调函数来解决这个问题，即函数作为参数传递到异步代码中，并在处理异常或处理“返回值”时立刻调用。Node.js 运行时非常依赖 JavaScript 的调度能力。为了确保异步错误能被处理以及传值正确，Node.js 的回调函数甚至有一套标准的函数签名。

不幸的是，异步代码嵌套多了就会变得很复杂，如清单 14-1 所示。

清单 14-1 异步 Node.js 例子

```
// example-001/index.js
'use strict';
var fs = require('fs');
var path = require('path');
var playerStats = require('./player-stats');

function getPlayerStats(gamesFilePath, playerId, cb) {
  // fs.readFile() 是异步函数
  fs.readFile(gamesFilePath, {encoding: 'utf8'}, function (err, content) {
    if (err) {
      return cb(err);
    }

    var games = JSON.parse(content);
    var playerGames = games.filter(function (game) {
      return game.player === playerId;
    });

    // playerStats.calcBest() 是异步函数
    playerStats.calcBest(playerGames, function (err, bestStats) {
      if (err) {
        return cb(err);
      }

      // playerStats.calcAvg() 是异步函数
      playerStats.calcAvg(playerGames, function (err, avgStats) {
        if (err) {
          return cb(err);
        }
        cb(null, {best: bestStats, avg: avgStats});
      });
    });
  });
}

var gamesFilePath = path.join(__dirname, 'games.json');
getPlayerStats(gamesFilePath, 42, function (err, stats) {
  if (err) {
    console.error(err);
    return process.exit(1);
  }
  console.log('best:', stats.best);
  console.log('avg:', stats.avg);
});
```

在这个例子中，JavaScript 代码分四步调用：

1. getPlayerStats() 的声明和调用。
2. fs.readFile() 的调用。
3. playerStats.calcBest() 的调用。
4. playerStats.calcAvg() 的调用。

很明显，playerStat 作为一个外部服务响应查询很慢。但如果是同步代码，如清单 14-2 所示，每个函数和方法都会按顺序被调用一次，try/catch 中的代码能处理同步代码生成的错误，异常原因也能输出到控制台。

清单 14-2 同步 Node.js 例子

```
// example-002/index.js
'use strict';
```



```

var fs = require('fs');
var path = require('path');
var playerStats = require('./player-stats');

try {
  var gamesFilePath = path.join(__dirname, 'games.json');
  // fs.readFileSync() 是同步函数
  var content = fs.readFileSync(gamesFilePath, {encoding: 'utf8'});
  var games = JSON.parse(content);
  var playerGames = games.filter(function (game) {
    return game.player === 42;
  });
  // playerStats.calcBestSync() 是同步函数
  console.log('best:', playerStats.calcBestSync(playerGames));
  // playerStats.calcAvgSync() 是同步函数
  console.log('avg:', playerStats.calcAvgSync(playerGames));
} catch (e) {
  console.error(e);
  process.exit(1);
}

```

这个同步代码的例子很容易完整地追踪每个语句块的执行流。异步回调驱动的代码在缓解这个问题的同时，还有许多显著的问题。

首先，回调函数签名不是必须遵从权威标准。虽然 Node.js 规范已被广泛采纳，但组建作者创建 API 时可不一定遵守这些规范。这通常发生在组建作者模仿一个（其他语言中）现有的、而 JavaScript 中没有的 API 时。组建作者可能会倾向于模仿该 API 现有的使用方式，而不遵循 Node.js 惯例。

其次，回调函数传递出错信息是手动的，即每个回调函数都必须检查 `err` 对象，并据此决定做什么还是转发给另一个做相同操作的回调函数。大量的错误检查样板代码都是手动检查 `err`。而在同步代码中，异常会自动向堆栈顶部扩散，直到被 `catch` 块捕获。

再次，很容易遗漏或错误地处理异步代码中的同步错误。在清单 14-3 中，`try/catch` 包裹着一个同步的 `JSON.parse` 调用，当解析成功时给回调传解析后的 JavaScript 对象，否则捕获异常。

清单 14-3 在 `try/catch` 中不恰当的回调函数调用

```

// example-003/improper-async-error-handling.js
'use strict';
var fs = require('fs');
var path = require('path');

function readJSONFile(filePath, cb) {
  fs.readFile(filePath, function (err, buffer) {
    try {
      var json = JSON.parse(buffer.toString());
      cb(null, json);
    } catch (e) {
      console.log('where did this error come from?', e.message);
      cb(e);
    }
  });
}

var gamesFilePath = path.join(__dirname, 'games.json');
readJSONFile(gamesFilePath, function (err, json) {
  if (err) {
    return console.error('parsing json did not succeed :(');
  }
  console.log('parsing json was successful :));

```

```
    throw new Error('should never happen, right?');
  });
```

假设 `games.json` 文件存在并且数据是合法的 JSON 格式，那么清单 14-3 所示的回调函数会在 `try` 块里调用，然后解析数据。但请注意当回调函数抛出异常后所发生的事情，这时回调函数产生的异常会展开堆栈回到 `try` 块，导致 `catch` 块捕获异常，此时回调函数被调用了两次。这很可能带来意想不到的后果。正确地处理这种错误方法，如清单 14-4 所示，避免同时在 `try/catch` 块中调用回调函数。

清单 14-4 在 `try/catch` 中正确地调用回调函数

```
// example-003/proper-async-error-handling.js
'use strict';
var fs = require('fs');
var path = require('path');

function readJSONFile(filePath, cb) {
  fs.readFile(filePath, function (err, buffer) {
    var json, err;
    try {
      json = JSON.parse(buffer.toString());
    } catch (e) {
      err = e;
    }
    if (err) {
      console.log('where did this error come from?', e.message);
      return cb(err);
    }
    cb(null, json);
  });
}

var gamesFilePath = path.join(__dirname, 'games.json');
readJSONFile(gamesFilePath, function (err, json) {
  if (err) {
    return console.error('parsing json did not succeed :(');
  }
  console.log('parsing json was successful :)' );
  throw new Error('should never happen, right?');
});
```

最后，嵌套的回调函数是一把双刃剑。一方面，每个回调函数都能访问各自所在闭包的数据，而闭包也包含着其回调函数；另一方面，回调函数嵌套会迅速产生复杂的、高度耦合的代码，使程序执行流变得模糊，从而导致系统维护性降低，容易产生 bug。

14.2 Promise 对比回调函数

为了解决异步回调带来的问题，JavaScript 社区成员提出了许多方案和规范，最终采用了 Promises/A+ 规范。这个规范以及其他衍生的规范，都定义了一种叫“Promise”的特殊对象来封装异步操作。多个 Promise 对象可以级联起来形成一种异步调用链，它们的值和错误信息可以通过链式调用传递并在需要的地方做处理。

根据 Promises/A+ 规范，一个 Promise 对象有 3 种状态来表现其异步操作：pending, fulfilled 和 rejected。从回调的角度看这些状态，则相当于清单 14-5。

清单 14-5 回调过程对应的 Promise 状态

```
// 调用这个函数等效于异步操作的状态为 "pending"
asyncFunction(function asyncCallback (err, asyncData) {
  if (err) {
    // 如果产生了错误，对应的状态就是 "rejected"
  }
  // 否则，异步操作的状态就是 "fulfilled"
});
```

Promise 因其 `then()` 方法也被称为“thenable”，`then()` 方法接收两个可选的回调函数参数：如果 Promise 的异步操作状态是 `fulfilled`，则第一个函数被调用；如果状态是 `rejected`，则第二个函数被调用。清单 14-6 展示了 `then()` 方法的完整函数签名。

清单 14-6 `then()` 方法的签名

```
/**
 * @param {Function} [onFulfilled]
 * @param {Function} [onRejected]
 * @returns {Promise}
 */
promise.then(onFulfilled, onRejected)
```

等等！Promise 的主要目的不是消除回调吗？不是，Promise 主要目的是为了let异步操作通过链式调用而变得简单，这个过程通常需要开发者展开嵌套的回调。注意，Promise 的 `then()` 方法实际上也返回了一个 Promise 对象；根据原始 Promise 对象回调函数的处理，`then` 返回的 Promise 状态也是 `fulfilled` 和 `rejected` 中的一个。利用这个特点，我们在清单 14-7 中重写了获取 `player` 状态的代码，这几乎消除了回调嵌套。

清单 14-7 Promise 减少嵌套

```
// example-004/index.js
'use strict';
var fs = require('fs');
var path = require('path');
var playerStats = require('./player-stats');
var Q = require('q');

function getPlayerStats(gamesFilePath, playerId, cb) {
  // load() 返回一个 Promise 对象
  playerStats.load(gamesFilePath, playerId)
    .then(function (games) {
      // Q.all() 返回一个 Promise 对象
      return Q.all([
        // calcBest() 返回一个 Promise 对象
        playerStats.calcBest(games),
        // calcAvg() 返回一个 Promise 对象
        playerStats.calcAvg(games)
      ]);
    })
    .done(function (allStats) {
      cb(null, {best: allStats[0], avg: allStats[1]});
    }, function (err) {
      cb(err);
    });
}

var gamesFilePath = path.join(__dirname, 'games.json');
getPlayerStats(gamesFilePath, 42, function (err, stats) {
  if (err) {
```

```

    console.error(err);
    return process.exit(1);
  }
  console.log('best:', stats.best);
  console.log('avg:', stats.avg)
});

```

在清单 14-7 中, 只有最后的 `done()` 调用接收了 `resolution` 和 `rejection` 回调, 其他所有 `then()` 只接收了一个 `resolution` 回调 (`playerStates` 模块中的函数)。这种连续调用 `thenable` 对象被称为 `Promise` 链。如果这些中间的某个 `then()` 调用过程中产生错误, 会发生什么呢? 跟异步回调模型不同, `Promise` 会通过 `Promise` 链向后自动传递异常, 直到异常被处理 (有点像模式化异常处理)。当然, 有特殊的规则和方法来改变这种行为, 但通常它的默认行为就非常符合大家的期望。

我会在稍后解释这个示例中其他有趣的地方 (例如, `fulfillment` 和 `rejection` 回调的返回值如何影响到整个 `Promise` 链)。很明显, `Promise` 能减少回调函数嵌套并自动传递错误, 这解决了异步 JavaScript 代码产生的两大问题。

14.3 Q 的 Promise

`Q` 是一个实现了 `Promises/A+` 规范的开源 JavaScript 库, 但它绝不是开发者的唯一选择。其他的几个库如 `when.js` 和 `Bluebird` 也提供了 `thenable` 对象。需要强调的是, 这个规范的既定目标是提供“所有符合 `Promises/A+` 规范的 `Promise` 实现可以相互依赖、相互操作的基础”。这意味着任何遵从规范的 `Promise` 库可以和其他相似的库混合使用, 开发者不用在一系列库中做出选择。大多数 `Promise` 库提供了一些辅助的功能来补充核心的 `thenable` 接口。开发者根据需要可自由选择、组合 `Promise` 库来解决不同的问题 (然而, 许多库并没有遵守 `Promises/A+` 规范, 如 `jQuery.Deferred` 就没有整合这个规范)。

本章使用 `Q` 作为主题的依据有几点:

1. 它遵从 `Promises/A+` 规范。
2. 它的作者 `Kris Kowal` 是规范的制定者之一。
3. 它在 JavaScript 社区被广泛采用和支持 (客户端和服务端)。
4. `Google` 的一款流行的浏览器 JavaScript 框架 `AngularJS`, 也大量借鉴 `Q`。

下面, 我们将用 `Promise` 替换异步回调驱动代码实现的例子来检验 `Q` 的实现。

14.3.1 Deferreds 和 Promises

虽然 `Promises/A+` 规范定义了 `thenable` 对象的行为, 但它没有明确说明一个异步操作应该怎样通过触发回调函数来提供 `thenable` 对象。它只定义了一些呈现 `Promise` 中异步操作状态的规则, 以及如何通过 `Promise` 链传递值和错误。实际上, 许多 `Promise` 库使用 `deferred` 对象来管理 `Promise` 状态。通常是在异步操作的开头先创建 `deferred` 对象, 然后生成一个 `Promise` 对象给后面的调用代码。清单 14-8 展示了如何创建一个 `deferred` 对象并返回 `Promise` 对象。

清单 14-8 创建一个 `deferred` 对象

```

var Q = require('q');

function asyncFunc() {

```



```

// 创建一个 deferred
var d = Q.defer();

// 执行一些异步操作并管理 *deferred*
// 返回 deferred 的 Promise 对象
return d.promise;
}

// 函数返回了一个 thenable 对象
asyncFunc().then(function () {
  // 成功 :)
}, function () {
  // 失败 :(
});

```

在清单 14-8 中，调用 `Q.defer()` 来创建一个 `deferred` 对象，该对象会在异步代码实际运行时处理 `Promise` 状态。重点是 `deferred` 拥有一个 `Promise` —— `asyncFunc()` 函数返回的对象，调用它的 `then()` 方法时可附上回调函数。`asyncFunc()` 实际的执行，以及订阅其返回的 `Promise` 状态变化都被放在了一起。但 `asyncFunc()` 是 `resolve` 还是 `reject` 它的 `deferred` 对象（改变返回的 `Promise` 状态）完全取决于开发者。

清单 14-9 简单实现了之前虚构的 `playerStats` 模块中的函数 `calcAvg()`。用归约操作对一组数字求和，然后除以个数（得出平均数）是同步操作。为了将其变成异步的，使用 `Node.js` 函数 `process.nextTick()` 包装代码，使代码在事件循环的下次迭代中运行（等效于使用 `setTimeout()` 或 `setImmediate()`）。如果计算成功，`d.resolve()` 就会将 `Promise` 状态置为 `resolved`，同时 will 一些值传递给 `Promise` 的 `resolution` 回调函数。如果出错（例如，游戏数组长度为 0，会产生除以 0 的错误），`Promise` 状态就会置为 `rejected` 并调用 `d.reject()` 回调函数。

清单 14-9 在 `calcAvg()` 的实现中使用 `deferred`

```

// example-004/player-stats.js
var Q = require('q');

module.exports = {
  // load: function (gamesFilePath, playerId) {...}

  // calcBest: function (games) {...},
  calcAvg: function (games) {
    var stats = {
      totalRounds: 0,
      avgRoundsWon: 0,
      avgRoundsLost: 0
    };

    var deferred = Q.defer();

    process.nextTick(function () {
      if (games.length === 0) {
        deferred.reject(new Error('no games'));
        return;
      }

      var wins = 0, losses = 0;
      games.forEach(function (game) {
        if (game.rounds === 0) return;
        stats.totalRounds += game.rounds;
        wins += game.won;

```

```

    losses += game.lost;
  });

  stats.avgRoundsWon = (wins / stats.totalRounds * 100)
    .toFixed(2) + '%';
  stats.avgRoundsLost = (losses / stats.totalRounds * 100)
    .toFixed(2) + '%';

  deferred.resolve(stats);
});

return deferred.promise;
}
};

```

清单 14-10 展示了如何使用 deferred 和 Promise 封装异步回调驱动接口。

清单 14-10 使用 deferred 和 Promise 封装异步回调驱动接口

```

// example-005/callbackdb/database.js
'use strict';

module.exports = {
  customer: {
    // requires a callback
    find: function (criteria, cb) {
      cb(null, {
        id: criteria.id,
        name: 'Nicholas Cloud'
      });
    }
  }
};

// example-005/callbackdb/find-customer-callback.js
var Q = require('q'),
    db = require('./database');

function loadCustomer(customerID) {
  var d = Q.defer();

  // db.customer.find() 是异步的
  db.customer.find({id: customerID}, function (err, customer) {
    if (err) {
      return d.reject(err);
    }
    d.resolve(customer);
  });

  return d.promise;
}

loadCustomer(1001).then(function (customer) {
  console.log('found', customer.id, customer.name);
}, function (err) {
  console.error(err);
});

```

这个封装异步代码的模型很普通。实际上，Q 提供了许多便捷的方法来减少写这种样板代码的负担。Q 的 deferred 对象有一个 `makeNodeResolver()` 方法，调用它的时候会创建一个虚拟的回调函数，这个函数可以传递任何标准的异步回调函数。当这个回调函数被调用时，根据适当的值或错误来改变 deferred 状

态，并把值或错误传给 Promise 回调函数。清单 14-11 展示了怎么用 resolver 替代手动编写的回调函数。

清单 14-11 使用 Node Resolver 回调函数

```
// example-005/callbackdb/database.js
'use strict';

module.exports = {
  customer: {
    // requires a callback
    find: function (criteria, cb) {
      cb(null, {
        id: criteria.id,
        name: 'Nicholas Cloud'
      });
    }
  }
};

// example-005/callbackdb/find-customer-makenoderesolver.js
var Q = require('q'),
    db = require('./database');
function loadCustomer(customerID) {
  var d = Q.defer();

  // db.customer.find() 是异步的
  var deferredCallback = d.makeNodeResolver();
  db.customer.find({id: customerID}, deferredCallback);

  return d.promise;
}

loadCustomer(2001).then(function (customer) {
  console.log('found', customer.id, customer.name);
}, function (err) {
  console.error(err);
});
```

在这个例子中，调用代码期望调用 loadCustomer() 返回的 Promise，而数据库 API 期望传一个回调函数，自然 makeNodeResolver() 很适合。相反也成立——如果调用代码 loadCustomer() 传入一个回调函数，而数据库 API 实际返回一个 Promise——那么调用 Promise 的 nodeify() 方法来通知回调函数就再好不过了。清单 14-12 中，Promise 就准确地运用了这一点。

清单 14-12 用 nodeify() 给 Promise 传递一个普通的异步回调函数

```
// example-005/promisedb/database.js
'use strict';
var Q = require('q');

module.exports = {
  customer: {
    // 返回一个 Promise 而不是用回调函数
    find: function (criteria) {
      return Q({
        id: criteria.id,
        name: 'Nicholas Cloud'
      });
    }
  }
};

// example-005/promisedb/find-customer-nodeify.js
```

```

var Q = require('q'),
    db = require('./database');

function loadCustomer(customerID, cb) {
  // db.customer.find() 返回一个 Promise
  db.customer.find({id: customerID})
    .nodeify(cb);
  /* equivalent to:
  * db.customer.find({id: customerID}).then(function (customer) {
  *   cb(null, customer);
  * }, function (err) {
  *   cb(err);
  * });
  */
}

loadCustomer(3001, function (err, customer) {
  if (err) {
    return console.err(err);
  }
  console.log('found', customer.id, customer.name);
});

```

14.3.2 值和错误

使用简单的值或错误信息 `resolve deferred` 能满足大部分需求，但是 `Promises/A+` 还定义了更多的 `resolve` 规则，并且 `Q` 也实现了，这让开发者能更好地控制 `Promise` 状态。

1. 用 `Promise` 值 `Resolve Deferred` 对象

`deferred` 的行为可以根据 `resolve()` 方法的传值不同而改变。如果值是一个正常的对象或元数据，那么就会被传递给 `deferred Promise` 的 `resolve` 回调函数。如果“值”是另一个 `Promise`，如清单 14-13 所示，则第二个 `Promise` 的状态完全“转发”给第一个 `Promise` 的回调函数；如果第二个 `Promise` 状态是 `resolved`，那么其值就会传递给第一个 `Promise` 的 `resolve` 回调函数；如果是 `rejected`，它的错误信息就会传递给第一个 `Promise` 的 `reject` 回调函数。

清单 14-13 给 `deferred` 对象的 `resolve` 传递 `Promise` 对象

```

// example-006/index.js
'use strict';

var Q = require('q'),
    airport = require('./airport'),
    reservation = require('./reservation');

function findAvailableSeats(departingFlights) {
  var d = Q.defer();
  process.nextTick(function () {
    var availableSeats = [];
    departingFlights.forEach(function (flight) {
      var openFlightSeats = reservation.findOpenSeats(flight);
      availableSeats = availableSeats.concat(openFlightSeats);
    });
  });
}

```



```

// 给 resolve 方法传递对象
if (availableSeats.length) {
  d.resolve(availableSeats);
} else {
  d.reject(new Error('sorry, no seats available'));
}
});
return d.promise;
}

function lookupFlights(fromAirport, toAirport, departingAt) {
  var d = Q.defer();
  process.nextTick(function () {
    var departingFlights = airport.findFlights(
      fromAirport, toAirport, departingAt
    );
    // 给 resolve 方法传递另一个 Promise 对象
    d.resolve(findAvailableSeats(departingFlights));
  });
  return d.promise;
}

lookupFlights('STL', 'DFW', '2015-01-10').then(function (seats) {
  console.log('available seats:', seats);
}, function (err) {
  console.error('sorry:', err);
});

```

因为第一个 deferred 的状态基本上依赖第二个 Promise 对象的 resolve 或 reject 回调函数，所以它的 pending 状态时间跟第二个 Promise 的保持一致。一旦第二个 Promise 对象被 resolve 或 reject，deferred 对象也变成相应的状态，并且调用对应的回调函数。

2. 在回调中转发值、错误和 Promise 对象

当 resolve 或 reject 回调函数收到值或错误时，会发生几种情况。如果是 Promise 链末尾（或没有级联的 Promise）的回调函数，实际代码通常会利用值来做一些事或者记录错误。

当调用 thenable 对象的 then() 方法时总是返回一个新的 Promise 对象，但我们可以借助 resolve 和 reject 回调函数来处理值和错误，并发送给新的 Promise，供后面的回调函数操作。

处理值很简单，仅仅是传给 resolve 回调函数修改或者变换后再返回。清单 14-14 中，数据库 Promise 在 resolve 回调函数中处理一个数组，然后返回经过适当过滤后的值。

清单 14-14 在 resolve 回调函数中返回一个值

```

// example-007/index.js
'use strict';
var db = require('./database');
function findPalindromeNames() {
  // db.customers.find() 返回一个 Promise 对象
  return db.customer.find().then(function (customers) {
    // 返回一个过滤后的数组，并且发送给下一个 resolve 回调
    return customers.filter(function (customer) {
      // 过滤掉不是回文名字的 customer
      var name = customer.name.toLowerCase();
      var eman = name.split('').reverse().join('');
      return name === eman;
    }).map(function (customer) {

```

```

    // 值返回 customer.name
    return customer.name;
  });
});
}

findPalindromeNames().then(function (names) {
  console.log(names);
});

```

resolve 回调函数也可以把错误向 Promise 链的后端传播，并且下一个 reject 回调函数会因为返回的错误而被调用。在清单 14-15 中，如果一个用户提交了太多次猜测（如一些猜测比赛），就会在 thenable 对象的 resolve 回调函数内部抛出错误。这个错误会传递给 Promise 链的下一个 reject 回调函数。

清单 14-15 在 resolve 回调函数中抛出一个错误

```

// example-008/index.js
'use strict';
var db = require('./database');
var MaxGuessError = require('./max-guess-error');

var MAX_GUESSES = 5;

function submitGuess(userID, guess) {
  // db.user.find() 返回一个 Promise
  return db.user.find({id: userID}).then(function (user) {
    if (user.guesses.length === MAX_GUESSES) {
      throw new MaxGuessError(MAX_GUESSES);
    }
    // 否则跟新用户信息...
  });
}

submitGuess(1001, 'Professor Plum').then(function () {
  // 如果报错就不会执行到这儿
  console.log('guess submitted');
}, function (maxGuessError) {
  // 出错啦！
  console.error('invalid guess');
  console.error(maxGuessError.toString());
});

```

回想刚刚的例子，如果是常用的异步回调模型，抛出的错误必须手动处理并统计错误数量（这意味着通常会遗漏一些不可预测的错误）。而 Q 处理错误是自动的，即任何在 thenable 内部抛出的异常都会被捕获并及时传递开——即使 thenable 回调被异步执行。

reject 回调函数也遵从一些相似的规则，但有许多区别。它们接受错误而非值，因此开发者理所应当以为 reject 回调会触发 Promise 链的下一个 reject 回调。但并不是！在清单 14-16 里，Promise 链的最后一个 Promise 会被 resolve，而不是被 reject——即使 submitGuess() 内部的 reject 回调函数返回的是一个错误。

清单 14-16 在 reject 回调函数中返回错误

```

// example-009/index.js
'use strict';
var db = require('./database');

```

```

var NotFoundError = require('./not-found-error');

function submitGuess(userID, guess) {
  // db.user.find() 返回一个 Promise
  return db.user.find({id: userID}).then(function (user) {
    /*
     * 数据库出错，因此这个 Promise 不会被 resolve。
     */
  }, function (err) {
    var notFoundError = new NotFoundError(userID);
    notFoundError.innerError = err;
    return notFoundError;
  });
}

submitGuess(1001, 'Colonel Mustard').then(function (value) {
  /*
   * 这个 Promise 被 resolve 了，并且
   * value === notFoundError!
   */
  console.log('guess submitted');
  console.log(value);
}, function (notFoundError) {
  /*
   * 如果你认为这个 Promise reject 了...
   * 那么你错了。
   */
  console.error('an error occurred');
  console.error(notFoundError);
});

```

这看起来和我们的直觉相反。如果 reject 回调函数返回错误，通常以为它会被传递，其实不会。再一看，其实是有道理的，因为这允许开发者处理一些不需要传递的错误并根据一些返回值优雅地 resolve Promise 链。

如果清单 14-16 的代码能在数据库不可用时将 guesses 加入到一个队列，那么在发生错误时，就能使后续 Promise 的 resolve 变得有意义，如清单 14-17 所示。

清单 14-17 在 reject 回调中隐藏错误信息

```

// example-010/index.js
'use strict';
var db = require('./database');
var guessQueue = require('./guess-queue');

function submitGuess(userID, guess) {
  // db.user.find() returns a Promise
  return db.user.find({id: userID}).then(function (user) {
    /*
     * 数据库出错，因此这个 Promise 不会被 resolve。
     */
  }, function (err) {
    console.error(err);
    /*
     * 数据库连接可能断了，将 guess 加入队列以后处理
     */
    return guessQueue.enqueue(userID, guess);
  });
}

submitGuess(1001, 'Miss Scarlett').then(function (value) {

```

```

/*
 * 当数据库连接失败, guess 已加入队列, 因此错误传递被抑制了。
 */
console.log('guess submitted');
}, function (notFoundError) {
  console.error('an error occurred');
  console.error(notFoundError);
});

```

跟 resolve 回调类似, 为了使下一个 Promise 状态正确地变为 rejected, reject 回调必须 throw 错误, 如清单 14-18 所示。

清单 14-18 在 reject 回调中 throw 错误

```

// example-011/index.js
'use strict';
var db = require('./database');
var NotFoundError = require('./not-found-error');

function submitGuess(userID, guess) {
  // db.user.find() returns a Promise
  return db.user.find({id: userID}).then(function (user) {
    /*
     * 数据库连接可能断了, 将 guess 加入队列以后处理
     */
  }, function (err) {
    /*
     * 错误被 *thrown*, 而不是 return
     */
    var notFoundError = new NotFoundError(userID);
    notFoundError.innerError = err;
    throw notFoundError;
  });
}

submitGuess(1001, 'Mrs. Peacock').then(function (value) {
  /*
   * 因为上一个 Promise 中错误是被 thrown 的, 所以这个 Promise 不会被 resolve
   */
}, function (notFoundError) {
  /*
   * Promise 是 rejected 状态
   */
  console.error('an error occurred');
  console.error(notFoundError);
});

```

就像 deferred 能被其他 Promise resolve 一样, thenable 在 resolved 或者 rejected 时也可以返回 Promise, 并且影响后续 Promise 的状态。reject 回调和 resolve 回调都能返回 Promise。清单 14-19 中, 当数据库调用成功时返回第二个 Promise, 否则抛出一个异常。

清单 14-19 resolve 回调返回另一个 Promise

```

// example-012/index.js
'use strict';
var db = require('./database');
var MaxGuessError = require('./max-guess-error');

```



```

var MAX_GUESSES = 5;

function submitGuess(userID, guess) {
  // db.user.find() 返回 Promise
  return db.user.find({id: userID}).then(function (user) {
    if (user.guesses.length === MAX_GUESSES) {
      throw new MaxGuessError(MAX_GUESSES);
    }
    // 更新 user
    user.guesses.push(guess);
    return db.user.update(user);
  });
}

submitGuess(1001, 'Professor Plum').then(function () {
  /*
   * 当数据库完成更新 user 操作就会调用此回调
   */
  console.log('guess submitted');
}, function (maxGuessError) {
  console.error('invalid guess');
  console.error(maxGuessError.toString());
});

```

3. 简单值转为 Promise 对象

Q 能将任何值转为 Promise，只需要简单地将 Q 作为函数调用，并将值作为参数传给它即可。清单 14-20 中，用 Q 包装了一个简单的字符串，并将其作为值传递给 Promise 链的下一个 Promise resolve 回调。

清单 14-20 将一个值转为 Promise

```

// example-013/index.js
'use strict';
var Q = require('q');

Q('khan!').then(function (value) {
  console.log(value); //khan!
});

```

这看起来再平凡不过了，但它作为一个基础的 API 能提供简单的方式将已存在的值或同步代码实际返回的值包装成 Promise 对象。如果调用 Q 时不传值，这会创建一个空的 Promise 对象，其状态为 resolved。

调用 Q 包装其他库的 Promise 对象也能得到 Q 的 Promise 对象，并能正常地调用 Q 的方法。这对开发者很有用。当开发者使用其他 Promise 库不能达到效果时，则可以使用一些 Q 的 Promise 方法如 nodeify() 来弥补。

14.3.3 报告进度

有时异步操作执行完需要花很长的时间。这段时间可以给调用代码一些进度报告，进度可以是一个简单的计量（如完成百分比），也可以报告可用数据的字节数。Q 扩展了 Promises/A+ 协议，为 then() 增加了第三个回调参数，可以捕获事件发生的进度，如清单 14-21 所示。

清单 14-21 Q 的 thenable 函数签名

```
/**
 * @param {Function} [onFulfilled]
 * @param {Function} [onRejected]
 * @param {Function} [onProgress]
 * @returns {Promise}
 */
promise.then(onFulfilled, onRejected, onProgress)
```

尽管 Promises/A+ 规范并没有建立进度通知模式，但是 Q 依然符合规范，因为其 thenable 都支持约定的 then() 函数签名。

正如当某个 deferred 变为 fulfilled 或 rejected 时会调用 fulfillment 和 rejection 回调一样，当 deferred 的 notify() 方法被调用时会执行 progress 回调。notify() 只接受一个参数，并且将其传给 progress 回调。在清单 14-22 中，一个长时运行的异步操作持续记录其完成一些操作的尝试次数（如调用几个常常反应迟钝的 API）。每做一次尝试都会使计数器增加，并将其值传给 notify() 方法。progress 回调会立即收到这个值。一旦 deferred 被 resolve 了，Promise 链也就结束了，并且调用最终的 done() 回调。

清单 14-22 通知 Deferred Promise 进度

```
<!-- example-014/index.html -->
<form>
  <p>The UI thread should respond to text field input, even though many DOM elements are
  being added.</p>
  <input type="text" placeholder="type something here" />
</form>
<div id="output"></div>

<script>
(function () {
  var Q = window.Q;
  var output = document.querySelector('#output');

  function writeOutput(msg) {
    var pre = document.createElement('pre');
    pre.innerHTML = msg;
    output.insertBefore(pre, output.firstChild);
  }

  function longAsync() {
    var d = Q.defer();

    var attempts = 0;

    var handle = setInterval(function () {
      // 每次异步代码执行时都会将 attempts 作为进度发送
      attempts += 1;
      d.notify(attempts);
      if (attempts === 1200) {
        clearInterval(handle);
        return d.resolve();
      }
    }, 0);

    return d.promise;
  }
  // 只传了 resolve 和 progress 回调，没有传 reject 回调
```

```

    longAsync().then(function () {
      writeOutput('done');
    }, null, function (attempts) {
      writeOutput('notification: ' + attempts);
    });
  }());
</script>

```

需要注意的是，尽管任何一个 thenable 的 resolution 或 rejection 回调都会根据 Promise 链的变化执行，但 progress 回调必须在通知事件执行之前附加到 thenable 上才能收到更新。思考清单 14-23 中的代码，其结果输出在清单 14-24 中。

清单 14-23 在添加 Progress 回调之前通知 Deferred 的 Promise

```

// example-015/index.js
'use strict';
var Q = require('q');

function brokenPromise() {
  var d = Q.defer();
  process.nextTick(function () {
    console.log('scheduled first');
    d.notify('notifying');
    d.resolve('resolving');
    console.log('logging');
  });
  return d.promise;
}

var Promise = brokenPromise();

process.nextTick(function () {
  console.log('scheduled second');
  promise.then(function (value) {
    console.log(value);
  }, null, function (progress) {
    console.log(progress);
  });
});

```

在清单 14-23 中，deferred 创建后，异步调用了 notify() 和 resolve()，这两个 deferred 的方法执行后才为 Promise then() 附加回调函数。清单 14-24 中的输出结果是代码作为 Node.js 脚本运行的结果。

清单 14-24 打印没有通知结果的输出

```

$ node index.js
scheduled first
logging
scheduled second
resolving

```

日志在 notify 或 resolve 之前就开始显示了，在 brokenPromise() 函数中代码实际运行时，还没有回调函数附加到 deferred 的 Promise then()。当运行到 then() 时，progress 回调函数已被完全忽略——尽管 resolution 回调收到了它的值。这是因为添加 progress 回调的时机放在了 deferred notify() 被调用之后。根据 Promises/A+ 规范，当新的回调附加到 thenable 时，为了保证值传递，resolutions 和 rejections 可以接收到值。但是 Q 将通知作为“实时”事件对待，因此必须在传值给回调函数之前附加 progress 回调函数。

14.3.4 终点

为了进一步模拟写同步代码的习惯，Q 提供了与同步代码中模式化异常处理特性相同的 `catch()` 和 `finally()` 方法。

`catch()` 方法其实是 `then(null, onRejection)` 的一个别名。`catch()` 允许开发者在 Promise 链的任何节点处理 error 而不会中断 Promise 链。清单 14-25 的代码中，使用 `catch()` 拦截可能出现 HTTP 请求失败的结果。因为 `catch()` 返回一个 Promise，所以它的回调可以返回给后续 Promise 需要的任何值（或抛出另一个 error）。

清单 14-25 Promise 链中 catch error

```
// example-016/index.js
'use strict';
var Q = require('q');
var api = require('./api');
var InvalidTeamError = require('./invalid-team-error');

function loadTeamRoster(teamID) {
  // api.get() returns a Promise
  return api.get('/team/' + teamID + '/roster')
    .catch(function (err) {
      /*
       * throw 一个有意义的异常，而不是传递 HTTP 错误
       */
      if (err.statusCode === 404) {
        throw new InvalidTeamError(teamID);
      }
    });
}

loadTeamRoster(123).then(function (roster) {
  console.log(roster);
}).catch(function (err) {
  console.error(err.message);
});
```

`finally()` 方法的行为跟 `then()` 相似，但有一点需要注意：它可能不会改变它接收到的任何值和错误，但它可能返回一个全新的 Promise 传给后续的 Promise 链。如果它返回空，那么它接收的最初的值和错误就不会传递了。

`finally()` 方法的真正目的是为了模拟 `try/catch/finally` 块的最后一部分。它允许代码在执行线程继续之前清理资源。清单 14-26 展示了在 `finally()` 块中如何关闭一个数据库连接。不管连接或更新操作是否执行成功，`finally()` 回调中的代码都会执行，且如果数据库保持连接打开，就清除关掉数据库连接。

清单 14-26 在 Promise 链中释放清理资源

```
// example-017/index.js
'use strict';
var Q = require('q');
var db = require('./database');

var user = {
  id: 1001,
```



```

    name: 'Nicholas Cloud',
    occupation: 'Code Monkey'
  });

db.connect().then(function (conn) {
  return conn.user.update(user)
    .finally(function () {
      if (conn.isOpen) {
        conn.close();
      }
    });
});

```

调用 `finally()` 并不会真的中断 Promise 链。但你有可能希望在代码某处中断 Promise 链，比如在许多异步操作执行后需要处理结束值和 `error`。有很多方式可以中断 Promise 链，最暴力的方式是直接忽略最后一个 `then()` 返回的 Promise。但这个 Promise 可能被之前的 Promise 链代码 `reject` 了，这样 Q 就会保留 Promise 链产生的任何错误，传递给后面添加的 `rejection` 回调。如清单 14-27 所示，如果 Promise 链中断后没有 `rejection` 回调，那么 `error` 就会凭空消失，不被处理，而后面的 `resolution` 回调也不会被执行到。

清单 14-27 错误地终止一个 Promise 链

```

// example-018/index01.js
'use strict';
var Q = require('q');

function crankyFunction() {
  var d = Q.defer();
  process.nextTick(function () {
    d.reject(new Error('get off my lawn!'));
  });
  return d.promise;
}

// 没有 rejection 回调来显示错误
crankyFunction().then(function (value) {
  console.log('never resolved');
});

```

为了解决这个问题，Q Promise 还有一个 `done()` 方法。它不返回 Promise，也不抛出任何错误，未处理的错误在下次事件循环时通过其他办法解决。其用法如清单 14-28 所示。

清单 14-28 使用 `done()` 终止 Promise 链

```

// example-018/index02.js
crankyFunction().done(function (value) {
  //...
});

```

虽然它没有 `rejection` 回调，但因为 `done()` 方法自动 `throw` 了一个错误，JavaScript 上下文也会终止 Promise 链。清单 14-29 中的输出打印，展示了使用 `done()` 方法终止 `crankyFunction()` Promise 链的结果。

清单 14-29 `done()` 抛出未处理的错误

```

$ node index02.js
/.../node_modules/q/q.js:126
  throw e;
    ^
Error: get off my lawn!

```

```

at /.../code/q/example-018/index02.js:7:14
at process._tickCallback (node.js:419:13)
at Function.Module.runMain (module.js:499:11)
at startup (node.js:119:16)
at node.js:906:3

```

14.4 控制流

Promise 链完美替代基于异步回调接口。它能模拟模式化异常处理模型，构建开发者熟悉的同步代码。这些特点简化了基于 Promise 代码的流程控制，但稍加创新，将多个异步操作分组并作为一个整体对待，就能利用 Promise 处理更复杂的流程。

- 顺序流：每个单独的异步操作按顺序依次执行，每个操作都在前一个操作完成后开始。
- 平行流：每个单独的异步操作同时执行并聚合所有的返回结果。
- 管道流：相互依赖的异步操作依次执行，每个操作执行都依赖上一个操作的返回值。

在这几种流程中，一般是异步操作的 rejection 回调触发流程失败。顺序流跟副作用相关，与值无关（它实际上不为后续操作提供数据）——即使它能根据需要聚合数据。平行流中聚合所有异步操作的数据，并在都执行完成的时候发送聚合的数据。管道流在异步操作之间传递数据，因此至少有一个异步操作会获取或创建数据，并在流的尾部处理这些值。

14.4.1 顺序流

清单 14-30 中是 Web 应用中常见的几个操作。它们一起完成修改用户的密码。当然，每个步骤都简化了很多，但以下三个基础步骤必须按顺序完成。

1. 改变密码。
2. 通知用户（通过邮箱等方式）他们的密码已被修改。
3. 因为我们的企业是良心企业，所以还会把密码转发给 National Security Agency (NSA)。

清单 14-30 函数按照顺序流执行

```

// example-019/index.js
function changePassword(cb) {
  process.nextTick(function () {
    console.log('changing password...');
    cb(null);
  });
}

function notifyUser(cb) {
  process.nextTick(function () {
    console.log('notifying user...');
    var randomFail = Date.now() % 2 === 0;
    cb(randomFail ? new Error('fail!') : null);
  });
}

function sendToNSA(cb) {
  process.nextTick(function () {
    console.log('sending to NSA...');
    cb(null);
  });
}

```

每个操作的函数都是异步执行，并且符合标准的 Node.js 回调模式。清单 14-30 中的 `changePassword()` 和 `sendToNSA()` 函数总会返回成功。为了使整个过程变得有趣，我们将 `notifyUser()` 函数的回调值经过计算，使其有时传正确的值、有时传错误的值。

为了整合这三个操作为顺序执行的 Promise 流，首先要将它们按照适当的执行顺序放在数组 “steps” 中。然后调用 Q 创建一个空的 Promise `lastPromise`，并且成为顺序执行的 Promise 链中第一个被 resolve 的 Promise。

清单 14-31 的代码中，遍历 `steps` 数组将每个异步操作封装为 Promise。每次迭代时调用 `lastPromise` 的 `then()` 方法，将返回的新的 Promise 赋给 `lastPromise` 变量（这是通过循环构建一个 Promise 链）。

在每个 resolution 回调中，通过 `Q.denodeify()` 将当前 “step”（清单 14-30 中定义的函数）转换为 Promise。当然也可以用清单 14-11 的方法手动完成，即创建一个 deferred 对象，并调用 `deferred.makeNodeResolver()`，但 `Q.denodeify()` 使这个过程更简单高效。resolution 回调返回这个 Promise，并作为 Promise 链的下一步骤。

清单 14-31 用 Promise 整合一个顺序流

```
// example-019/index.js
var Q = require('q');

var steps = [changePassword, notifyUser, sendToNSA];
var lastPromise = Q();
steps.forEach(function (step) {
  lastPromise = lastPromise.then(function () {
    /*
     * denodeify and invoke each function step
     * to return a Promise
     */
    return Q.denodeify(step)();
  });
});

lastPromise.done(function () {
  console.log('all done');
}, function (err) {
  console.error(err);
});
```

最终 resolution 和 rejection 回调都被添加到最后的 Promise 上。

当下一次事件循环执行时，第一个 step 就开始执行。当它 resolve 时，下一个 Promise 就会被执行，以此类推，直到顺序流的末尾。如果中间某个地方出错，就会直接导致最终的 rejection 回调被执行（因为这中间没有 rejection 回调；顺序流的某个步骤失败，整个过程都应该失败）。如果每一步的异步操作都 resolve 了，那么最终的 resolution 回调就会输出打印消息：all done。

14.4.2 平行流

在平行流中，应用程序通常从多个不同的源获取数据，然后将数据作为一个整体发送给调用代码。清单 14-32 中，当网页上的用户修改自己的邮寄地址时，就需要同时获得用户信息和美国州市列表。

清单 14-32 函数按照平行流执行

```
// example-20/index01.js
function getUser(id, cb) {
```

```
process.nextTick(function () {
  cb(null, {id: id, name: 'nick'});
});
}

function getUSStates(cb) {
  process.nextTick(function () {
    cb(null, ['MO', 'IL' /*, etc.*/]);
  });
}
```

因为这两个异步函数之间没有相互影响，所以让它们并行执行（而不是等待另一个执行完成）是合理的。Q 的实用函数 `all()` 可将一个数组的 Promise 设置为并行执行。但清单 14-32 中的函数还不是 Promise，必须调用 Q 的函数来转换。因为这些函数符合 Node.js 回调规范，所以可以像清单 14-33 那样用 `Q.nfcall()`（调用 node 函数）把函数包装成 Promise，使用 `deferred` 提供合适的回调函数。`getUser()` 函数接收一个数值参数。当为 `getUser()` 创建 Promise 时，必须将 `userId` 作为第二个参数传递给 `Q.nfcall()`。当内部调用 `getUser()` 时，Q 会把 `userId` 绑定到 `getUser()` 的第一个参数。

Q 的 `all()` 方法也返回一个 Promise，并使用一个数组值 `resolve` 这个 Promise。每个值的顺序跟传入 `Q.all()` 的 Promise 顺序一致。在这个例子中，用户信息是数组的第一个元素，美国州市列表是第二个元素。

如果某个 Promise 出错，那么 `all` 返回的 Promise 的 `rejection` 回调就会被执行。

清单 14-33 将 Promise 按照平行流组织

```
// example-20/index01.js
var Q = require('q');

Q.all([
  Q.nfcall(getUser, 123),
  Q.nfcall(getUSStates)
]).then(function (results) {
  console.log('user:', results[0]);
  console.log('states:', results[1]);
}, function (err) {
  console.error('ERR', err);
});
```

通过数组访问值其实是不雅的，因此 Q 提供了 `spread()` 方法。它能将结果数组“展开”为真实独立的函数参数，并跟 `then()` 方法的使用一致，如清单 14-34 所示。

清单 14-34 展开结果数组

```
// example-20/index02.js
var Q = require('q');

Q.all([
  Q.nfcall(getUser, 123),
  Q.nfcall(getUSStates)
]).spread(function (user, states) {
  console.log('user:', user);
  console.log('states:', states);
}, function (err) {
  console.error('ERR', err);
});
```


Q 还提供了一个 `all()` 的伴侣函数 `Q.allSettled()`，它们的行为很相似，只有几个不同的地方：首先，它始终会调用总的 Promise 的 resolution 回调；其次，每个 Promise 的值都是一个对象，并且包含 `state` 属性，用来报告该值对应的 Promise 的实际状态。以下属性也依赖 `state` 的值。

- 如果 Promise 是 resolved 状态，`value` 属性的值就是 Promise 产生的数据。
- 如果 Promise 是 rejected 状态，`reason` 属性的值就是 Promise 产生的错误信息。

选择 `Q.all()` 还是 `Q.allSettled()` 需要根据实际情况而定，但是两者都可以创建平行流。

14.4.3 管道流

当一组数据需要按照一些规则顺序地变换时，管道流就很有用了。管道流跟之前介绍的顺序流的差别是管道流需要给下一步传递数据，而顺序流则关注创建的一系列线性副作用。

清单 14-35 呈现的是一个简单的过滤系统，就像一个招聘系统，寻找客户需要的人才。`loadCandidates()` 函数会“获取”一份候选人名单，其他函数负责根据一些条件踢出部分候选人。`filterBySkill()` 和 `groupByStates()` 其实是工厂函数，它们接收一些配置参数（技能以及规定的需求），然后返回函数，这个函数接收一个 Node.js 回调函数，并被用到管道流中。

清单 14-35 函数按照管道流执行

```
// example-021/index.js
function loadCandidates(cb) {
  console.log('loadCandidates', arguments);
  process.nextTick(function () {
    var candidates = [
      {name: 'Nick', skills: ['JavaScript', 'PHP'], state: 'MO'},
      {name: 'Tim', skills: ['JavaScript', 'PHP'], state: 'TN'}
    ];
    cb(null, candidates);
  });
}

function filterBySkill(skill) {
  return function (candidates, cb) {
    console.log('filterBySkill', arguments);
    candidates = candidates.filter(function (c) {
      return c.skills.indexOf(skill) >= 0;
    });
    cb(null, candidates);
  };
}

function groupByStates(states) {
  var grouped = {};
  states.forEach(function (state) {
    grouped[state] = [];
  });
  return function (candidates, cb) {
    console.log('groupByStates', arguments);
    process.nextTick(function () {
      candidates.forEach(function (c) {
        if (grouped.hasOwnProperty(c.state)) {
          grouped[c.state].push(c);
        }
      });
      cb(null, grouped);
    });
  };
}
```

`loadCandidates()` 函数被直接添加到过程步骤数组，而 `filterBySkill()` 和 `groupByStates()` 函数按照初始值执行后将返回值放入步骤数组。

管道流也使用 Promise 链，设置执行顺序。清单 14-36 中，每步产生的结果值都会传递到对应的 Promise 的 resolution 回调，在 resolution 回调中将结果值放入一个数组，然后作为参数传递给执行顺序的下一个 Promise。在平行流的例子中，使用 `Q.nfcall` 封装每个异步操作，在这个例子中则使用 `Q.nfapply()` (`node-function-apply`)。它们的调用模仿了原生 JavaScript 函数 `Function.prototype.call()` 和 `Function.prototype.apply()`，这就是为什么结果值不是直接传递而是放在数组中传递。这很有必要，因为管道流的第一步 `loadCandidates()` 不接收任何参数（不同于其他回调），为了确保 `Q.nfapply()` 调用正常，所以传了一个空的数组。

清单 14-36 用 Promise 实现管道流

```
// example-021/index.js
var Q = require('q');

var steps = [
  loadCandidates,
  filterBySkill('JavaScript'),
  groupByStates(['MO', 'IL'])
];
var lastPromise = Q();
steps.forEach(function (step) {
  lastPromise = lastPromise.then(function (result) {
    var args = [];
    if (result !== undefined) {
      args.push(result);
    }
    return Q.nfapply(step, args);
  });
});

lastPromise.done(function (grouped) {
  console.log('grouped:', grouped);
}, function (err) {
  console.error(err);
});
```

在管道流末尾，传给最后一个异步回调的值，经过处理后会传给 `done()` 的 resolution 回调。如果任何一个异步操作产生了错误，都会调用 rejection 回调。

清单 14-35 中是单个值传递给各异步操作的回调。虽然 Promises/A+ 规范规定只允许传递一个值作为 resolution 回调的参数，但还是可以想办法传递多个值。Q 为了缓和与规范的冲突，将多个值打包成一个数组传递给异步操作，然后传递给各个 Promise 的 resolution 回调。这个数组需要通过 `Q.nfapply()`，将它包含的数据用作下一步骤的参数。

14.5 小结

回调是处理异步代码的标准机制。它为开发者提供了一些控制流技巧，使得在未来的事件循环完成之后能继续执行自己的代码。但是回调很容易使代码嵌套、耦合难以维护。

使用 Q 这样的 Promise 库封装异步操作，可以消除嵌套，提高代码的质量。Q 是开发者的一把利器，它可以自动传递值和错误信息、以异步方式级联回调函数、在长时异步操作运行期间报告

执行进度以及在 Promise 链的末端进行错误处理等。

Q 可以控制简单的、线性的程序流。适当创新，Q 还可以处理一些更复杂的程序流。这一章检验了 Q 实现顺序流、平行流和管道流，而且 Q 的实用方法能让开发者更灵活地实现其他程序流。

14.6 相关资源

- Q: <https://github.com/krisKowal/q>
- Promises/A+ spec: <https://promisesaplus.com/>

的一系列步骤。同样的例子还会在第 16 章的顺序流中用到，具体步骤可能会稍有不同。

首先，每个操作都会被包裹成一个工厂函数：接受初始数据为参数，并返回一个基于回调的函数，作为顺序流中的一步。

第一步 changePassword 任务从会会把所获取的新密码凭证作为参数传给它的回调。顺序流中的每一步相互独立，并不要求前一步要有执行结果。如果执行结果传给它的回调，也不会对后续步骤产生影响。回调函数可以返回一个 Promise 对象，以便后续步骤等待。本章会带大家从 Async 函数和 Promise 控制流开始。

本章会带大家从 Async 函数和 Promise 控制流开始。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

使用 Promise 解决上述问题。使用 Promise 解决上述问题。使用 Promise 解决上述问题。

Async.js

万事总有出其不意，但永远相信美好的事情即将发生。

——罗伯特·乔丹

软件流程控制很麻烦，尤其是涉及异步逻辑的时候。第 16 章会展示如何用 `promise` 解决这一难题。而本章会带大家认识 `Async.js`——一套回调驱动的 JavaScript 函数库，它提供了许多强大的异步处理和流程控制函数。

第 16 章会覆盖 3 种常见的异步流程问题：顺序、并行和管线，同时展示如何使用 `Q` 提供的辅助函数，将所有回调封装成 `promise`，使用 `promise` 解决上述问题。而 `Async.js` 则继续使用回调，采用异步编程模式，避免嵌套回调带来的诸多问题。

`Async.js` 的流程控制函数，大部分遵循以下范式。

1. 第一个参数通常是一个包含待执行任务函数的数组。这些数组中的任务函数根据流程控制函数的不同会有些区别，不过最后一个参数一定是一个 `Node.js` 风格的回调。
2. 最后一个参数同样是个回调函数，在所有任务函数执行完之后触发。该回调函数也接收一个 `Node.js` 风格的回调函数作为参数，作为参数的回调函数可传入或不传入参数。

注意 `Node.js` 风格的回调会一直将 `error` 作为第一个参数，同时附带其他值作为参数。当回调函数被调用时，`error` 要么是一个错误对象，要么是 `null`。清单 15-1 演示如何应用流程控制。

清单 15-1 流程控制函数范式

```
var tasks = [
  function (*0..n args, /* cb) { /*...*/ },
  function (*0..n args, /* cb) { /*...*/ },
  function (*0..n args, /* cb) { /*...*/ }
];
```

```
function finalCallback (err, result) { /*...*/ };
```

```
14 async.someFlowControlFunction(tasks, finalCallback);
```

本章以下内容会介绍更多流程控制函数，比较各函数的异同点，但是这些流程控制函数在任务函数处理、异常处理、值的返回等方面都遵循上述范式。这样设计也更容易理解和使用。

注意 `async` 在 `Async.js` 中的含义主要指组织异步操作。函数库本身不保证内部任务函数是否以异步来执行。如果开发人员使用 `Async.js` 处理同步任务，则每个任务会同步执行。但是也有特殊情

况，`async.memoize()`函数虽然与流控制无关，但可以让一个任务函数缓存自身结果，以便后续调用时不需要实际调用任务函数，而是使用其缓存的执行结果。`Async.js` 会强制后续的每次调用都是异步的，因为它假定最初的任务函数是异步的。

15.1 顺序流

顺序流即按顺序执行一系列任务。除序列里的第一个任务外，其他任务必须等到之前的任务完成之后才能启动，并且如果任何一个任务失败，则整体流程会失败。清单 15-2 旨在演示修改密码时的一系列步骤，同样的例子还会在第 16 章的顺序流中用到，具体步骤可能会稍有不同。

首先，每个操作都会被包裹成一个工厂函数，接受初始数据为参数，并返回一个基于回调的函数，作为顺序流中的一步。

其次，第一步 `changePassword()` 任务结束会把所获取的新密码凭证作为参数传给它的回调。顺序流中的每一步相互独立，并不要求前一步要有执行结果。如果执行结果传递给它的回调，也不会对其他步骤产生影响。如果某一步的执行依赖于之前步骤的执行结果，则应当使用管线流。（管线流将在稍后讨论。）

清单 15-2 顺序的步骤

```
// example-001/async-series.js
'use strict';
var async = require('async');
var userService = require('./user-service');
var emailService = require('./email-service');
var nothingToSeeHere = require('./nothing-to-see-here');
function changePassword(email, password) {
  return function (cb) {
    process.nextTick(function () {
      userService.changePassword(email, password, function (err, hash) {
        // 返回新密码凭证作为结果
        cb(null, {email: email, passwordHash: hash});
      });
    });
  };
}

function notifyUser(email) {
  return function (cb) {
    process.nextTick(function () {
      // 调用 Email 服务，无返回结果
      emailService.notifyPasswordChanged(email, cb);
    });
  };
}

function sendToNSA(email, password) {
  return function (cb) {
    process.nextTick(function () {
      // 调用 nothingToSeeHere 服务，无返回结果
      nothingToSeeHere.snoop(email, password, cb);
    });
  };
}
```

清单 15-3 中，每个工厂函数初始化并执行，将返回的任务函数添加到任务序列数组中。这个数

组会作为 `async.series()` 的第一个参数，第二个参数是一个回调函数，接受任务序列执行过程中的异常 `error`，或者执行结果的结果数组。结果数组中的每个元素与对应任务在任务序列中的位置一一对应。例如，`changePassword()` 的执行结果是 `results` 数组的第一个元素，因为 `changePassword()` 是任务序列的第一个任务。

清单 15-3 修改密码的步骤

```
// example-001/async-series.js
var email = 'user@domain.com';
var password = 'foo!1';

var steps = [
  //返回 function(cb)
  changePassword(email, password),
  //返回 function(cb)
  notifyUser(email),
  //返回 function(cb)
  sendToNSA(email, password)
];
async.series(steps, function (err, results) {
  if (err) {
    return console.error(err);
  }
  console.log('new credentials:', results[0]);
});
```

这些任务都是异步的，它们本身无法像同步方法一样依次调用并执行。但是 `Async.js` 会跟踪每个任务的内部执行，当且仅当前一个任务回调后再调用下一个任务，从而实现一个顺序的流式调用。如果该序列中任一步骤出现异常抛出错误信息，该序列将中止执行，并将该错误信息作为参数调用结果回调。此时，`results` 将是 `undefined`。

本章中使用的工厂函数可以很方便地将初始数据传给每个步骤，但它们并非必需的，可以使用清单 15-4 中 JavaScript 原生的 `bind` 机制来替代。对于没有初始数据的简单任务，也可以直接在数组中使用匿名函数。（不过还是出于可读性和可维护性的角度建议使用具名函数。）

清单 15-4 参数绑定的系列步骤

```
function changePassword(email, password, cb) { /*...*/ }

function notifyUser(email, cb) { /*...*/ }

function sendToNSA(email, password, cb) { /*...*/ }

var steps = [
  changePassword.bind(null, email, password),
  notifyUser.bind(null, email),
  sendToNSA.bind(null, email, password)
];
```

在本章的剩余部分，我们将使用工厂函数，而不是 `bind()`。不过开发者可以自由选择适合的任一方法。

15.2 并行流

有时我们希望并行执行所有的任务，在全部执行完毕后，聚合所有的结果再做进一步处理。

JavaScript 是异步语言（因为 JavaScript 是单线程执行的——译者注），它没有真正的并行，但在调度连续执行较长的非阻塞操作时，会使用事件循环处理其他操作（如在浏览器环境下的 UI 更新或者服务器环境下的处理更多请求）。一次事件循环中可以调度处理多个异步任务，但是仍然无法预测每个任务会在何时执行完成。这使得难以收集来自每个任务的调用结果，并返回给调用方。幸运的是，`async.parallel()` 函数可以满足开发人员的这一需求。

清单 15-5 是两个封装的 jQuery GET 请求。第一个请求会根据 `userID` 获取用户信息，第二个请求美国各州的列表。不难猜出这些功能是用户详细信息页面的一部分，用户能够在这个页面更新一些个人信息，如手机号、邮寄地址等。当页面加载时，需要一次性获取所有信息。虽然这些信息由两个不同的 API 调用，但是无论是否同时被调用，未来却会同时需要这两个请求的结果做下一步处理。

清单 15-5 并行的步骤

```
// example-002/views/async-parallel.html
function getUser(userID) {
  return function (cb) {
    $.get('/user/' + userID).then(function (user) {
      cb(null, user);
    }).fail(cb);
  };
}

function getUSStates(cb) {
  $.get('/us-states').then(function (states) {
    cb(null, states);
  }).fail(cb);
}
```

清单 15-6 中，`Async.js` 会在页面中注入一个标准的 `<script>` 标签。任务被 `async.parallel()` 函数调度，与 `async.series()` 参数类似，一个是任务序列数组，一个是接受 `error` 或者汇总结果的回调。并行任务只有一个回调函数作为参数，须在该任务执行结束后调用。所有回调符合 Node.js 的回调约定。

清单 15-6 的 `getUser()` 是个工厂函数，接受 `userID` 作为参数，返回一个符合 Node.js 回调约定的函数。因为 `getUSState()` 不需要任何参数，所以这里无需封装，直接使用。

这两个函数都使用 jQuery 的 AJAX API。AJAX 使用 `promise` 特性，在请求成功后，调用 `then()` 里传入的回调函数；当失败或出错时，调用 `fail()` 中传入的回调函数。因为 `fail()` 中传入的回调函数也接受一个单独的 `error` 参数，所以可以直接复用 `Async.js` 传给每个任务的回调。

清单 15-6 并行流

```
<!-- example-002/views/async-parallel.html -->
<h1>User Profile</h1>
<form>
  <fieldset>
    <div>
      <label>First Name</label>
      <input type="text" id="first-name" />
    </div>
    <div>
      <label>US States</label>
      <select id="us-states"></select>
    </div>
  </fieldset>
</form>
```

```

<script>
(function (async, $) {
    function getUser(userID) {
        return function (cb) {
            $.get('/user/' + userID).then(function (user) {
                cb(null, user);
            }).fail(cb);
        };
    }

    function getUSStates(cb) {
        $.get('/us-states').then(function (states) {
            cb(null, states);
        }).fail(cb);
    }

    var userID = 1001;

    async.parallel([
        getUser(userID),
        getUSStates
    ], function (err, results) {
        if (err) {
            return alert(err.message);
        }
        var user = results[0],
            states = results[1];
        $('#first-name').val(user.firstName);
        // ...
        $('#us-states').append(states.map(function (state) {
            return $('<option></option>')
                .html(state)
                .attr('value', state);
        }));
    });
})(window.async, window.jQuery));
</script>

```

遍历任务数组中每一项任务，并同时执行。每个任务完成返回的执行结果都被保存下来。一旦所有的任务都执行完成，就将缓存的结果传递给 `async.parallel()` 的结果回调。

结果按照传递给 `async.parallel()` 的任务的顺序进行排序，而不是执行快慢排序。如果并行任务中任何一个出现 `error`，`error` 会被传给最终结果回调。所有未执行的任务，在完成后的结果也会被丢弃。最终的结果回调中的 `results` 参数值将是 `undefined`。

15.3 管线流

当任务序列中的每个任务的执行依赖于前一个的结果时，就需要使用管线流（也称瀑布流）。清单 15-7 演示一个虚拟用户奖励计划，企业根据用户的年龄（利用出生日期计算）分配不同的奖金。任务序列中的每个任务接受一些输入，然后把输出传给回调函数，同时，前一个任务的输出会作为下一个任务的输入。

1. `getUser()` 接受 `userID` 作为输入参数，然后返回一个任务函数。该任务会查询数据库中的 `user` 记录，并把查询到的 `user` 结果作为参数传给它的回调。

2. `calcAge()` 会接受 `user` 信息，并计算出该用户的 `age`，作为参数传给它的回调。
3. `reward()` 接受数字类型的 `age` 参数，并匹配出对应的奖金，作为参数传给它的回调。

清单 15-7 瀑布（管线）的步骤

```
// example-003/callback-waterfall
'use strict';
var db = require('./database');

function getUser(userID, cb) {
  process.nextTick(function () {
    // pass cb directly to find because
    // it has the same signature:
    // (err, user)
    db.users.find({id: userID}, cb);
  });
}

function calcAge(user, cb) {
  process.nextTick(function () {
    var now = Date.now(),
        then = user.birthDate.getTime();
    var age = (now - then) / (1000 * 60 * 60 * 24 * 365);
    cb(null, Math.round(age));
  });
}

function reward(age, cb) {
  process.nextTick(function () {
    switch (age) {
      case 25: return cb(null, '$100');
      case 35: return cb(null, '$150');
      case 45: return cb(null, '$200');
      default: return cb(null, '$0');
    }
  });
}
```

如果使用嵌套回调的话，整个处理流程将会相当可怕。同时，如果有更多的任务加入这个奖励计划的话，代码需要分开处理，并进行大规模的重新组织，才能满足需求。而且如果中间出错的话，需要手动捕获并处理回调。清单 15-8 演示在不使用 `Async.js` 的情况下，代码是如何处理的。

清单 15-8 使用嵌套回调的瀑布流

```
// example-003/callback-waterfall
function showReward(userID, cb) {
  getUser(userID, function (err, user) {
    if (err) {
      return cb(err);
    }
    calcAge(user, function (err, age) {
      if (err) {
        return cb(err);
      }
      reward(age, cb);
    });
  });
}

showReward(123, function (err, reward) {
```

```

    if (err) {
      return console.error(err);
    }
    console.log(reward);
  });

```

幸运的是，有了 Async.js 后，我们可以相对轻松地组织管线流的代码，使其兼具可维护性和优雅的错误处理机制。清单 15-9 的代码使用 `async.waterfall()` 组织一系列的待执行任务，然后提供一个最终回调来捕获管线流中的错误，或者最终的 `reward` 执行结果。

清单 15-9 瀑布（管线）流

```

// example-003/async-waterfall.js
'use strict';
var async = require('async');
var db = require('./database');

```

```

function getUser(userID) {
  // using a factory function to pass in
  // the userID argument and return another
  // function that will match the callback
  // signature that async.waterfall expects
  return function (cb) {
    process.nextTick(function () {
      // pass cb directly to find because
      // it has the same signature:
      // (err, user)
      db.users.find({id: userID}, cb);
    });
  };
};

// the calcAge and reward functions
// do not change

```

```

async.waterfall([
  getUser(1000),
  calcAge,
  reward
], function (err, reward) {
  if (err) {
    return console.error(err);
  }
  console.log('reward:', reward);
});

```

与 `async.series()` 和 `async.parallel()` 类似，管线流中任意一个任务出现异常 `error`，整个任务会立即中止，并将 `error` 作为参数调用最终回调。

管线重用

管线对于数据处理十分方便。`async.seq()` 与 `async.waterfall()` 类似，接受一系列的任务函数，将它们合并成一个可以多次调用的可重用的管线函数。当然，这可以通过封装 `async.waterfall()` 手动实现，但是 `async.seq()` 帮助开发者避免了不必要的麻烦。

清单 15-10 中的程序展示了一系列处理手机账单的功能函数。`createBill()` 函数接受通话计划作为参数，基于该计划和正常月租生成 `bill` 账单对象。`carrierFee()` 会在之前的基础上追加部分邮递费用。之后，`prorate()` 再检测部分款项是否已经记入账户名下（如用户在计费周期中又开启了新的计划）。最后，`govtExtortion()` 还会在账单公布前追加相应的税款。

清单 15-10 序列（管线）的步骤

```
// example-004/async-seq.js
'use strict';
var async = require('async');
var dateUtil = require('./date-util');

function createBill(plan, cb) {
  process.nextTick(function () {
    var bill = {
      plan: plan,
      total: plan.billAmt
    };
    cb(null, bill);
  });
}

function carrierFee(bill, cb) {
  process.nextTick(function () {
    bill.total += 10;
    cb(null, bill);
  });
}

function prorate(bill, cb) {
  if (!bill.plan.isNew) {
    return cb(null, bill);
  }
  process.nextTick(function () {
    bill.plan.isNew = false;
    var days = dateUtil().daysInMonth();
    var amtPerDay = bill.plan.billAmt / days;
    var prorateAmt = ((bill.plan.billDay - 1) * amtPerDay);
    bill.total -= prorateAmt;
    cb(null, bill);
  });
}

function govtExtortion(bill, cb) {
  process.nextTick(function () {
    bill.total = bill.total * 1.08;
    cb(null, bill);
  });
}
```

如清单 15-11 所示，利用 `async.seq()` 创建管线与 `async.waterfall()` 是非常相似的。它们的主要区别在于，`async.seq()` 不会立即调用任务序列，而是返回一个 `pipeline()` 函数，再调用任务序列。`pipeline()` 函数直接使用第一步中传入的参数作为初始数据，避免了之前管线定义时，在第一步使用工厂函数或者数值绑定的麻烦。此外，与其他的 Async.js 的异步函数不同，`async.seq()` 是可变参数的（接受不同数量的参数）。`async.waterfall()` 接受任务数组作为参数，`async.seq()` 则接受每个任务函数作为参数。

清单 15-11 演示了 `pipeline()` 的创建及其参数的调用方式：一个 `plan` 对象，之后会作为参数传给 `createBill()`，以及最终的结果回调，接受 `error` 异常或者最终用户的 `bill` 账单对象。

清单 15-11 序列（管线）流

```
// example-004/async-seq.js
var pipeline = async.seq(
  createBill,
```

```

    carrierFee,
    prorated,
    govtExtortion
  );
};

var plan = {
  type: 'Lots of Cell Minutes Plan!+',
  isNew: true,
  billDay: 15,
  billAmt: 100
};
pipeline(plan, function (err, bill) {
  if (err) {
    return console.error(err);
  }
  //bill = govtExtortion(prorated(carrierFee(createBill(plan))))
  console.log('$', bill.total.toFixed(2));
});

```

15.4 循环流

不断重复执行直到某些条件得到满足，称为循环（loop）。Async.js 有多个循环函数，帮助协调在特定条件下执行相应的异步代码。

15.4.1 为真则循环执行

前两个函数，即 `async.whilst()` 与 `async.doWhilst()`，等同于我们所熟知的许多编程语言中的 `while` 和 `do/while` 循环模式。在某些条件成立时，循环得以执行。条件不再成立时，循环被终止。

`async.whilst()` 与 `async.doWhilst()` 几乎完全相同，除了 `async.whilst()` 会在执行循环体之前先进行条件判断，而 `async.doWhilst()` 则是先执行 1 次循环体中的代码，再进行条件判断。当条件不成立时，`async.whilst` 可能一次都不会被执行，而 `async.doWhilst()` 会至少执行一次。

清单 15-12 演示了 `async.whilst()` 如何被用于循环调用 10 次 API 接口，获取战斗中“随机”的获胜者。循环运行之前，会先构造一个数组，用来存放和监测是否已经挑选了 10 个获胜者。在数组长度达到 10 之前，循环体会被重复调用。当循环调用 API 的某一次出现 `error` 异常时，`async.whilst()` 会立即中止，最终结果回调会带着这个 `error` 被调用；否则，循环会持续执行直到条件不被满足，然后执行最终结果回调。

清单 15-12 为真则循环执行

```

<!-- example-005/views/async-whilst.html -->
<h1>Winners!</h1>
<ul id="winners"></ul>

<script>
(function (async, $) {

  function pickWinners(howMany, cb) {
    var winners = [];

    async.whilst(
      // condition test:
      // continue looping until we have enough winners

```



```

function () { return winners.length < howMany; },
// looping code
function (cb) {
  $.get('/employee/random').done(function (employee) {
    var winner = employee.firstName + ' ' + employee.lastName;
    // avoid potential duplicates
    if (winners.indexOf(winner) < 0) {
      winners.push(winner);
    }
    cb(null);
  }).fail(function (err) {
    cb(err);
  });
},
// final callback
function (err) {
  // if there is an error just ignore it
  // and pass back an empty array, otherwise
  // pass the winners
  cb(null, err ? [] : winners);
}
);
});

pickWinners(3, function (err, winners) {
  $('#ul#winners').append(winners.map(function (winner) {
    return $('<li></li>').html(winner);
  }));
});

})(window.async, window.jQuery));
</script>

```

清单 15-13 对清单 15-12 的代码做了简短的修改，使用 `async.doWhilst()` 替代 `async.whilst()`。需要注意参数的顺序也发生了变化。`async.doWhilst()` 的循环体是第一个参数，条件判断变成了第二个。其模式上与 `do/while` 语法一致。

清单 15-13 先执行一次之后，等待条件满足后再循环执行

```

<!-- example-005/views/async-dowhilst.html -->
<h1>Winners!</h1>
<ul id="winners"></ul>

<script>
(function (async, $) {

  function pickWinners(howMany, cb) {
    var winners = [];

    async.dowhilst(
      // looping code
      function (cb) {
        $.get('/employee/random').done(function (employee) {
          var winner = employee.firstName + ' ' + employee.lastName;
          // avoid potential duplicates
          if (winners.indexOf(winner) < 0) {
            winners.push(winner);
          }
          cb(null);
        }).fail(function (err) {
          cb(err);
        });
      },

```

```
// condition test is now the second function
// argument
function () { return winners.length < howMany; },
// final callback
function (err) {
  // if there is an error just ignore it
  // and pass back an empty array, otherwise
  // pass the winners
  cb(null, err ? [] : winners);
}
);
}
}

pickWinners(3, function (err, winners) {
  $('#ul#winners').append(winners.map(function (winner) {
    return $('<li></li>').html(winner);
  }));
}));

})(window.async, window.jQuery));
</script>
```

15.4.2 为假则循环执行

相近的两个函数 `async.until()` 与 `async.doUntil()`，都采用相似的运行方式，不过前者是为真则循环，后者是为假才循环。

清单 15-14 演示了一个简单的 HTTP 心跳链接，用来在浏览器终端上测试 API 的可用性。`HeartBeat()` 构造函数使用 `async.until()` 创建了一个循环，然后重复执行，直到 `isStopped` 属性变为 `true`。`HeartBeat()` 对外暴露了一个 `stop()` 方法，如果在对象实例化之后被调用，会阻止循环的继续执行。每循环一次都会向服务器发起一个 HTTP 请求，如果请求成功，则设置 `isAvailable` 为 `true`；如果失败，`isAvailable` 会置为 `false`。为了在每次循环中做下间隔，这里使用 `setTimeout` 延迟循环体的调用时机，从而达到调度并推迟（本例 3 秒钟）接下来循环体执行的目的。

清单 15-14 为假则循环执行

```
<!-- example-006/views/async-until.html -->
<section id="output"></section>
```

```
<script>
(function (async, $) {

  var output = document.querySelector('#output');

  function write() {
    var pre = document.createElement('pre');
    pre.innerHTML = Array.prototype.join.call(arguments, ' ');
    output.appendChild(pre);
  }

  function Heartbeat(url, interval) {
    var self = this;
    this.isAvailable = false;
    this.isStopped = false;
    this.writeStatus = function () {
      write(
        '> heartbeat [isAvailable: %s, isStopped: %s]'
      );
    };
  }
})(window.async, window.jQuery);
```

```

        .replace('%s', self.isAvailable)
        .replace('%s', self.isStopped)
    );
};

async.until(
    // test condition
    function () { return self.isStopped; },
    // loop
    function (cb) {
        $.get(url).then(function () {
            self.isAvailable = true;
        }).fail(function () {
            self.isAvailable = false;
        }).always(function () {
            self.writeStatus();
            // delay the next loop by scheduling
            // the callback invocation in the
            // future
            setTimeout(function () {
                cb(null);
            }, interval);
        });
    },
    // final callback
    function (/*err*/) {
        self.isAvailable = false;
        self.writeStatus();
    }
);

Heartbeat.prototype.stop = function () {
    this.isStopped = true;
};

var heartbeat = new Heartbeat('/heartbeat', 3000);

setTimeout(function () {
    // 10 seconds later
    heartbeat.stop();
}, 10000);

}(window.async, window.jQuery));
</script>

```

`async.doUntil()`函数的使用与 `async.doWhilst()`非常相近：先运行一次，再进行条件判断。其标志也是交换条件判断和循环体的参数顺序。

15.4.3 循环重试

循环使用中的一个常见场景是循环重试。当我们希望某些任务达到给定次数的时候，假如任务执行失败并且重试次数还没达到上限，它会重新执行；当它达到重试上限，则中止。`async.retry()`函数通过封装重试逻辑，帮助开发者简化了这一过程。建立一个循环变得很简单，只需要明确重试次数、重试的任务以及最终的回调函数，来处理错误或结果。

清单 15-15 演示了一个简单的为电影或演唱会预留座位的 API。可用座位存在一个数组中，最优

选的排在最前面。执行次数不能超过数组的长度。每次任务执行时，会把数组前面第一个（最优选的）位子从数组中拿掉。如果预留失败，会继续执行，直到没有更多的座位中止。

清单 15-15 循环重试

```
<!-- example-007/views/async-retry -->
<section id="output"></section>

<script>
(function (async, $) {

    var output = document.querySelector('#output');

    function write() {
        var pre = document.createElement('pre');
        pre.innerHTML = Array.prototype.join.call(arguments, '\n');
        output.appendChild(pre);
    }

    function reserve(name, availableSeats) {
        console.log(availableSeats);
        return function (cb) {
            var request = {
                name: name,
                seat: availableSeats.shift()
            };
            write('posting reservation', JSON.stringify(request));
            $.post('/reservation', request)
                .done(function (confirmation) {
                    write('confirmation', JSON.stringify(confirmation));
                    cb(null, confirmation);
                }).fail(function (err) {
                    cb(err);
                });
        };
    }

    var name = 'Nicholas';
    var availableSeats = ['15A', '22B', '13J', '32K'];

    async.retry(
        availableSeats.length,
        reserve(name, availableSeats),
        function (err, confirmation) {
            if (err) {
                return console.error(err);
            }
            console.log('seat reserved:', confirmation);
        }
    );
})(window.async, window.jQuery);
</script>
```

每次任务执行时会调用自己的回调。任务执行成功，会把结果传给回调，最终把返回的结果（本例中是 `confirmation`）传给 `async.retry()` 并进行调用。如果出现错误，循环会重复执行，直到达到重试次数限制。最后一个 `error` 会回传给最终回调，之前的会全部丢掉——除非手动保存。清单 15-16 演示了一个潜在方案，把错误收集到数组中，然后用 `err` 对象替换为该数组，传递给最终回调。如果重试循环失败的话，最终回调收到的 `error` 将会是由每次执行过程中返回的 `error` 组成的数组。

清单 15-16 重试循环中收集错误

```
function reserve(name, availableSeats) {
  var errors = [];
  return function (cb) {
    // ...
    $.post('/reservation', body)
      .done(function (confirmation) {
        cb(null, confirmation);
      }).fail(function (err) {
        errors.push(err);
        cb(errors);
      });
  };
}
```

15.4.4 无限循环

在同步编程模式中，无限循环因为会长期占用 CPU 资源、阻碍其他代码的运行，而被看作一个灾难。但是异步编程中的无限循环不会受上述问题影响，因为像所有其他的代码一样，它们都是被 JavaScript 的事件循环调度。可被植入于其他需要运行的代码中或者可被调度的请求。

`async.forever()` 函数可以用来调度生成一个无限循环。它接受任务函数作为第一个参数，最终回调函数作为第二个。它生成的任务将会无限期地调用——除非在某次调用中返回了异常。创建异步操作可以使用延迟时间为 0 的 `setTimeout()`，或者使用接近立即执行的 `setImmediate()`。所以最好使每个异步任务都有比较长的等待时间，至少几百毫秒。

清单 15-17 中的无线循环会在每个周期内发起一个 HTTP GET 请求，为用户面板拉取股票信息。每次 GET 请求成功，都会更新股票信息，然后在下一轮请求之前等待 3 秒钟。如果循环过程中出现 error，循环中止，并且该 error 作为参数触发最终回调。

清单 15-17 无限循环

```
<!-- example-008/views/async-forever.html -->
<ul id="stocks"></ul>
```

```
<script>
(function (async, $) {
  $stockList = $('ul#stocks');

  async.forever(function (cb) {
    $.get('/dashboard/stocks')
      .done(function (stocks) {
        // refresh the stock list with new stock
        // information
        $stockList.empty();
        $stockList.append(stocks.map(function (stock) {
          return $('<li></li>').html(stock.symbol + ' $' + stock.price);
        }));
        // wait three seconds before continuing
        setTimeout(function () {
          cb(null);
        }, 3000);
      }).fail(cb);
    }, function (err) {
      console.error(err.responseText);
    })
  })
```

```
}(window.async, window.jQuery));
</script>
```

15.5 批处理流

本章介绍的最后一个流控制类型是批处理流。批处理把数据分隔成块即为批次，然后同时进行操作。批处理限制了一个块中最多可存放多少数据。批处理开始处理队列中的一个数据块后，新的数据被加入批处理流中，直到处理完成，然后获取一个新的数据块再处理。

15.5.1 异步队列

异步队列是批处理流中的常用方法。调用 `async.queue()` 来创建异步队列，需要传递两个参数：第一个是函数，处理添加到队列中的每一个数据项；第二个是数字，设置队列中将同时进行数据处理任务的最大数量。清单 15-18 中的异步队列会向所有添加到队列中的 URL 发起 HTTP 请求。每个 HTTP 请求完成后的结果将添加到 `results` 散列中。最多可以同时进行的 HTTP 请求数是 3。当 3 个请求均在处理的时候又有新的 URL 加入队列中，它们会暂时挂起等待未来调用。当工作程序被释放（请求完成后），它们立即会被用来请求队列中等待的其他 URL。但同一时刻绝不会进行 3 个以上的 HTTP 请求。

清单 15-18 使用队列进行顺序的批处理

```
// example-009/index.js
'use strict';
var async = require('async');
var http = require('http');

var MAX_WORKERS = 3;
var results = {};

var queue = async.queue(function (url, cb) {
  results[url] = '';
  http.get(url, function (res) {
    results[url] = res.statusCode + ' Content-Type: ' + res.headers['content-type'];
    cb(null);
  }).on('error', function (err) {
    cb(err);
  });
}, MAX_WORKERS);

var urls = [ // 9 urls
  'http://www.appendto.com',
  'http://www.nodejs.org',
  'http://www.npmjs.org',
  'http://www.nicholascloud.com',
  'http://www.devlink.net',
  'http://javascriptweekly.com',
  'http://nodewebly.com',
  'http://www.reddit.com/r/javascript',
  'http://www.reddit.com/r/node'
];
urls.forEach(function (url) {
  queue.push(url, function (err) {
```

```

    if (err) {
      return console.error(err);
    }
    console.log('done processing', url);
  });
});

```

队列在其声明周期中的特定点会触发一系列事件。可以监听这些事件并绑定函数做一些特殊处理。事件处理函数为可选。

当队列达到设置的最大同时处理上限时，会调用 `queue.saturated` 上绑定的函数。但队列处理完所有项目且没有其他项目在排队时，会调用 `queue.empty` 函数。当所有执行进程都已经完成，并且队列为空时，会调用 `queue.drain`。清单 15-19 中的函数演示了监听并处理这些事件。

清单 15-19 队列事件

```

// example-009/index.js
queue.saturated = function () {
  console.log('queue is saturated at ' + queue.length());
};

queue.empty = function () {
  console.log('queue is empty; last task being handled');
};

queue.drain = function () {
  console.log('queue is drained; no more tasks to handle');
  Object.keys(results).forEach(function (url) {
    console.log(url, results[url]);
  });
  process.exit(0);
};

```

■ **注意** `empty` 与 `drain` 存在一些微妙的区别。当 `empty` 触发时，工作程序可能仍然在运行——虽然此时队列中已经没有待处理项目。当 `drain` 触发时，所有工作程序都已经停止，队列完全是空的。

15.5.2 异步负载

`async.cargo()` 函数与 `async.queue()` 比较相似，都会排队处理项目。它们的区别在于如何划分工作负载。`async.queue()` 限制并使用多个工作程序同时处理。而 `async.cargo()` 同一时间只会有一个工作程序在运转，但是根据预定大小，分割排队项目成多个载荷。一个载荷处理完后，处理下一个，直到全部完成。负载的阈值是每个载荷的大小。如果工作程序已经开始处理载荷，之后任何加入到负载的新项目会分组到下一个有效载荷中处理。

通过传入一个任务函数作为 `async.cargo()` 的第一个参数，传入一个最大的负载数目作为第二个参数来创建一个 `cargo`。任务函数接受一个待处理的数据数组（最大长度是负载大小），以及一个操作完成后的回调函数。

清单 15-20 演示了使用 `async.cargo()` 把一系列的数据库更新操作打包成一个虚构的交易，同一时间只有一个载荷。任务函数会遍历传入的 `update` 对象，把每个元素转换成一个 `UPDATE` 的虚拟关系型数据库操作。一旦更新操作都已完成，交易提交，触发最终的回调函数。

清单 15-20 使用 cargo 进行异步批量操作

```
// example-010/index-01.js
'use strict';
var async = require('async');
var db = require('db');

var MAX_PAYLOAD_SIZE = 4;
var UPDATE_QUERY = "UPDATE CUSTOMER SET ? = '?' WHERE id = ?";

var cargo = async.cargo(function (updates, cb) {
  db.begin(function (trx) {
    updates.forEach(function (update) {
      var query = UPDATE_QUERY.replace('?', update.field)
        .replace('?', update.value)
        .replace('?', update.id);
      trx.add(query);
    });
    trx.commit(cb);
  });
}, MAX_PAYLOAD_SIZE);

var customerUpdates = [ // 负载 4 中增加 9 个更新
  {id: 1000, field: 'firstName', value: 'Sterling'},
  {id: 1001, field: 'phoneNumber', value: '222-333-4444'},
  {id: 1002, field: 'email', value: 'archer@goodisis.com'},
  {id: 1003, field: 'dob', value: '01/22/1973'},
  {id: 1004, field: 'city', value: 'New York'},
  {id: 1005, field: 'occupation', value: 'Professional Troll'},
  {id: 1006, field: 'twitter', value: '@2cool4school'},
  {id: 1007, field: 'ssn', value: '111-22-3333'},
  {id: 1008, field: 'email', value: 'urmom@internet.com'},
  {id: 1009, field: 'pref', value: 'rememberme=false&colorscheme=dark'}
];

customerUpdates.forEach(function (update) {
  cargo.push(update, function () {
    console.log('done processing', update.id);
  });
});
```

如清单 15-21 所示, cargo 对象与 queue 对象具有相同的事件属性, 主要区别是当最大数目的负载加入以后, 会以有效载荷的阈值开始处理。

清单 15-21 cargo 事件

```
// example-010/index-01.js
cargo.saturated = function () {
  console.log('cargo is saturated at ' + cargo.length());
};

cargo.empty = function () {
  console.log('cargo is empty; worker needs tasks');
};

cargo.drain = function () {
  console.log('cargo is drained; no more tasks to handle');
};
```


■ **注意** 由 `async.queue()` 与 `async.cargo()` 调度的任务函数，都会在下一个事件循环中立即执行。如果项目以同步的方式一个接一个被添加到 `cargo` 或 `queue` 中，则会如预期执行：队列会限制一个最大的处理进程，`cargo` 会把项目分割成最大载荷来处理。如果项目以异步的方式加入——在下次事件循环中加入，任务函数可能会以小于最大容量的数目调用。

清单 15-22 中，将每一次更新操作从 `customerUpdates` 数组中拿出，将它放入大 `cargo` 中，每隔 500 毫秒重复执行一次。因为 `cargo` 会立即调度执行任务，所以每次会同时执行 1 到 2 个 `UPDATE` 操作，具体数目依赖于每个任务执行多久，以及 `cargo` 调度下一个任务的间隔时间。

```
清单 15-22 使用异步的方式添加项目到 Cargo 中
// example-010/index-02.js
(function addUpdateAsync() {
  if (!customerUpdates.length) return;
  console.log('adding update');
  var update = customerUpdates.shift();
  cargo.push(update, function () {
    console.log('done processing', update.id);
  });
  setTimeout(addUpdateAsync, 500);
})();
```

如果想充分达到 `cargo` 的处理阈值，建议使用同步的方式添加项目。

15.6 小结

本章覆盖了一些常用的同步控制流，并演示了如何使用 `Async.js` 将这些模式应用到异步代码上。表 15-1 展示了每种流类型和对应的 `Async.js` 的函数。

表 15-1 流和 Async.js 中对应的函数	
流	Async.js 函数
顺序	<code>async.series()</code>
并行	<code>async.parallel()</code>
管线	<code>async.waterfall()</code> , <code>async.seq()</code>
循环	<code>async.whilst()/async.dowhilst()</code> , <code>async.until()/async.doUntil()</code> , <code>async.retry()</code> , <code>async.forever()</code>
批处理	<code>async.queue()</code> , <code>async.cargo()</code>

顺序流和并行流允许开发者执行多个独立的任务，得到汇总结果。管线流可以用来将多个任务连接到一起，上个任务的输出变成下个任务的输入。循环流可以给定次数，或者给定限制条件，重复执行异步任务。最后，批处理流可用于将数据分割成块，分批执行。

通过巧妙地安排异步任务函数，协调各个任务的结果，将 `error` 或任务执行结果返回给最终回调，`Async.js` 帮助开发者有效避免了嵌套回调的陷阱，并将传统同步流控制的开发模式带到了异步 JavaScript 的世界。

Underscore 和 Lodash

你一定是那类能将事情做好的[人士]。但为了将事情做好，首先你得喜欢做这件事，而不是喜欢这件事情的结果，那仅仅是第二位。

——安·兰德

JavaScript 是种实用的工具型语言。因为简洁的 API 及松散的类型系统，它能应用在很多场合。JavaScript 的基本知识点很少，是种容易学习和掌握的语言。这种特性让它能有很高的生产效率，但这也意味着在历史上 JavaScript 的类型系统就缺乏一些高级特性，而这些特性能增强语言的健壮性，比如源自于集合以及哈希的函数迭代构造。

为了弥补这种差距，Jeremy Ashkenas 在 2009 年建立了一个名为 Underscore 的库，它涵盖超过 100 种用来操作、过滤、哈希变换与集合的函数集。这其中的很多诸如 `map()` 和 `reduce()` 之类的函数，都体现了与函数式编程语言相同的理念。而其他函数，比如 `isArguments()`、`isUndefined()` 则只特定于 JavaScript。

随着 Underscore 在许多 Web 应用程序中的广泛应用，发生了两件激动人心的事：第一，ECMAScript 5 标准于同年发布，原生 JavaScript 对象包含许多类似 Underscore 的方法，如 `Array.prototype.map()`、`Array.prototype.reduce()` 和 `Array.isArray()`，成为 ECMAScript 5 特色。但尽管 ECMAScript 5（和 ECMAScript 6/7 中较少程度地）扩展了一些 Underscore 的主要 API，它包含的 Underscore.js 功能还是很少。

第二，为了达到大幅提升性能及扩展 API 的目的，Underscore 被克隆（Git/GitHub 中仓库 fork）为一个叫作 Lodash 的新项目。由于 Lodash 不仅实现了 Underscore 中所有函数，还增加了其本身特有函数，所以说 Underscore 是 Lodash 的子集。此外，所有对应于 Underscore 的 ECMAScript 标准函数亦是 Lodash 的一部分。可以看到，表 16-1 展示了 Underscore 和 Lodash 与对应的原生 ECMAScript 的映射关系。

表 16-1 Underscore 和 Lodash 与目前（及提案）原生 JavaScript 对比

ECMAScript 5	Underscore/Lodash
<code>Array.prototype.every()</code>	<code>all()/every()</code>
<code>Array.prototype.filter()</code>	<code>select()/filter()</code>
<code>Array.prototype.forEach()</code>	<code>each()/forEach()</code>
<code>Array.isArray()</code>	<code>isArray()</code>
<code>Object.keys()</code>	<code>keys()</code>
<code>Array.prototype.map()</code>	<code>map()</code>

续表

ECMAScript 5	Underscore/Lodash
<code>Array.prototype.reduce()</code>	<code>inject()/foldl()/reduce()</code>
<code>Array.prototype.reduceRight()</code>	<code>foldr()/reduceRight()</code>
<code>Array.prototype.some()</code>	<code>some()</code>
ECMAScript 6	Underscore/Lodash
<code>Array.prototype.find()</code>	<code>find()</code>
<code>Array.prototype.findIndex()</code>	<code>findIndex()</code>
<code>Array.prototype.keys()</code>	<code>keys()</code>
ECMAScript 7	Underscore/Lodash
<code>Array.prototype.contains()</code>	<code>include()/contains()</code>

由于 Underscore 和 Lodash 共享同一套 API，因而 Lodash 可以用作 Underscore 的一种直接替代。但由于 Lodash 的额外功能，反过来则不一定行得通。例如，虽然 Underscore 和 Lodash 都具有 `clone()` 方法，却只有 Lodash 实现了 `deepClone()` 方法。开发者更愿意用 Lodash 而不是 Underscore，通常是因为这些额外函数，除此之外，还和 Lodash 明显的性能受益有关。通过函数间标准性能检查程序，Lodash 比 Underscore 平均快 35%。它通过优先使简单循环而不是用类似 `forEach()`、`map()`、`reduce()` 等原生函数代理，实现了明显的性能提升。

本章集中讨论 Lodash 和 Underscore 中还没被 ECMAScript 实现（或提案）的大部分特性（见清单 16-1 及清单 16-2 中的函数）。Mozilla 的优秀文档涵盖每个原生函数。同样，Underscore 和 Lodash 的文档中也涵盖其每个函数的实现。

Underscore 和 Lodash 提供了很多适合操作对象与集合的函数，而不是仅仅提供几个常用函数。本章将探索其中的部分函数。

注意 为简洁起见，本章的余下部分只引用了 Underscore，但请理解在没有特别指明的情况下，本章讨论的 Underscore 和 Lodash 都是可互换的。

16.1 安装及用法

在浏览器或任何服务器端 JavaScript 环境（如 Node.js）中，Underscore 均可以直接作为库导入，不需要外部依赖。

您可直接从 Underscore 官网（<http://underscorejs.org>）下载 Underscore.js 文件，或者通过类似 NPM、Bower、Component 的包管理器安装它。

在浏览器中，您可以将 Underscore 作为一个 script 资源直接引入，或者通过实现了 AMD 或 CommonJS 的模块加载器加载（如 RequireJS 或 Browserify）。在 Node.js 中，则只需简单地作为一个 CommonJS 模块引入 Underscore 组件。

访问 Underscore 对象（其工具函数所附属的对象）的方式取决于函数库的加载方式。当 Underscore 通过浏览器的 `script` 标签加载，库将自动附属在 `window._` 上。如清单 16-1 所示，对在一般环境下通过模块加载器创建的变量，可以把当前的 Underscore 模块直接赋值给字符，十分方便。

清单 16-1 从 Node.js 模块中加载 Underscore 库

```
// example-001/index.js
'use strict';
var _ = require('underscore');

console.log(_.VERSION);
// 1.8.2
```

所有 Underscore 函数都存在于 `_` (“下划线”) 对象上。由于 Underscore 是一个工具库，除了少量设置以外，它不会维持任何状态（本章节后面会涉及这些内容）。所有函数都是幂等 (*idempotent*) 的，也就是说，对任意函数多次传递同一个值，每次返回的结果都是相同的。一旦 Underscore 对象加载，就能立即使用它。

Underscore 的实用函数操作主要针对集合（数组和类数组对象，比如 `argument`）、对象字面量及函数。Underscore 最常用的场合是在过滤和数据变换上。许多 Underscore 函数之间都是相互补充的，这些函数互相作用在一起又能形成功能强大的组合。由于这种组合使用函数的方式会很常用，Underscore 就内置了函数链式调用的功能。键式调用功能创建了简洁的管线式操作，这种操作允许一次进行多个数据转换。

16.2 聚合和索引

集合中的数据经常共享类似模式，这些数据又具有标识属性，使得每个数据均为唯一。在一个数据集合中，为了迅速过滤、使用满足匹配聚合条件的对象子集，区分这两类关系——共性和个性则是十分有用的。

Underscore 具有若干方法来执行这类任务，但对处理集合来说，则有三个函数能发挥极大的作用：`countBy()`、`groupBy()`、`indexBy()`。

16.2.1 countBy()

对共享一类特征的对象进行计数是归纳数据的常用方式。对一个 URL 集合，可以设想若干分析过程来判定多少 URL 属于某顶级域名（如 `.com`、`.org`、`.edu` 等）。对于这个任务，Underscore 的 `countBy()` 函数可以说是个理想的选择。它会为数组中的每个元素调用一个回调函数，以决定该元素符合哪个类别（本例中即每个 URL 属于哪个顶层域名）。该回调函数的返回值为代表该类别的字符串值。程序最终得到的结果是一个对象，该对象的每个键代表由回调函数返回的每个类别，而该对象的值则是每个类别中所有相应元素的数量。清单 16-2 展示了上述过程的基本实现：其返回了一个对象，该对象中 `.org` 域名的数量为 2，`.com` 域名的数量为 1。

清单 16-2 按某条件对元素计数

```
// example-002/index.js
'use strict';
var _ = require('underscore');

var urls = [
  'http://underscorejs.org',
  'http://lodash.com',
  'http://ecmascript.org'
];

var counts = _.countBy(url, function byTLD(url) {
  if (url.indexOf('.com') >= 0) {
```



```

    return '.com';
  }
  if (url.indexOf('org') >= 0) {
    return '.org';
  }
  return '?';
});

console.log(counts);
// { '.org': 2, '.com': 1 }

```

如果一个集合中每一项都是拥有属性的对象，且某个特定属性的值代表需要被统计的数据，就不需要迭代器函数了。取而代之的是，可以把被检测属性的名称作为参数。注意，在清单 16-3 中，最终结果的每一个键都将会是在每个对象上得到检测的那个特定属性的值。

清单 16-3 按某属性对元素计数

```

//example-003/index.js
'use strict';
var _ = require('underscore');

var urls = [
  {scheme: 'http', host: 'underscore.js', domain: '.org'},
  {scheme: 'http', host: 'lodash', domain: '.com'},
  {scheme: 'http', host: 'ecmascript', domain: '.org'},
];

var counts = _.countBy(urls, 'domain');

console.log(counts);
// { '.org': 2, '.com': 1 }

```

若该集合中有一个或多个对象缺少需要被测试的属性，那么最终的结果对象就会包含一个为 `undefined` 的键，该键的值是集合中所有缺少该属性的对象总数。

16.2.2 groupBy()

`Underscore` 的 `groupBy()` 函数与 `countBy()` 相似，但不同于把结果归纳为数字形式的数量值，`groupBy()` 在结果对象中会把元素放入分类后的集合内。清单 16-4 中的 URL 对象都被放在每个与之对应的顶级域名集合中。

清单 16-4 按某属性为元素分组

```

// example-004/index.js
'use strict';
var _ = require('underscore');

var urls = [
  {scheme: 'http', host: 'underscorejs', domain: '.org'},
  {scheme: 'http', host: 'lodash', domain: '.com'},
  {scheme: 'http', host: 'ecmascript', domain: '.org'},
];

var groupd = _.groupBy(urls, 'domain');

console.log(groupd);

/*
{

```

```

    '.org': [
      { scheme: 'http', host: 'underscorejs', domain: '.org' },
      { scheme: 'http', host: 'ecmascript', domain: '.org' },
    ],
    '.com': [
      { scheme: 'http', host: 'lodash', domain: '.com' }
    ]
  }
}
*/

```

注意 在为元素分类时，如果需要对分类元素进行更大程度的控制，`groupBy()`函数的第二个参数也可换成迭代器函数（把属性名参数换掉）。

值得一提的是，仅需通过简单的查询分组后，对象中每个分组数组的长度、数量值就能很容易地推导出来。在依赖于应用程序上下文时，比起使用计数函数，选择使用分组函数会更实用。可以看到，清单 16-5 不仅展示了一个函数，该函数通过 `groupBy()` 得到的数量值结果建立了对象，还展示了如何从分组后的数据获取单一集合的数量值。

清单 16-5 从分组数据中推导出数量值

```

// example-005/index.js
'use strict';
var _ = require('underscore');

var urls = [
  { scheme: 'http', host: 'underscorejs', domain: '.org' },
  { scheme: 'http', host: 'lodash', domain: '.com' },
  { scheme: 'http', host: 'ecmascript', domain: '.org' },
];

var grouped = _.groupBy(urls, 'domain');
var dotOrgCount = grouped['.org'].length;
console.log(dotOrgCount);
// 2

function toCounts(grouped) {
  var counts = {};
  for (var key in grouped) {
    if (grouped.hasOwnProperty(key)) {
      counts[key] = grouped[key].length;
    }
  }
  return counts;
}

console.log(toCounts(grouped));
// { '.org': 2, '.com': 1 }

```

16.2.3 indexBy()

这个函数可以用于发现集合中数据的差异，尤其是当这些差异能作为唯一标识符时。通过已知标识符找出集合中的某个对象是相当普遍的场景。如果使用手动操作的方式实现这个功能，则需要遍历集合中每个元素（可以使用 `while` 或 `for`），并返回第一个被匹配到该唯一标识符的元素。可以想象这样一个场景：一个用于顾客选择出发地和目的地的航线网站，用户通过下拉菜单选择某个机场，就会显示出该个机场的附加信息。附加信息从数组中的机场对象加载。每个下拉菜单中被选中的值

是唯一机场编码，该编码用于后续获取机场对象的完整详细信息。所幸的是，本应用的开发者使用 Underscore 的 `indexBy()` 函数从机场数组中创建了一个索引对象，如清单 16-6 所示。

清单 16-6 通过属性为对象建立索引

```
// example-006/index.js
'use strict';
var _ = require('underscore');

var airports = [
  {code: 'STL', city: 'St Louis', timeZone: '-6:00'},
  {code: 'SEA', city: 'Seattle', timeZone: '-8:00'},
  {code: 'JFK', city: 'New York', timeZone: '-5:00'}
];

var selected = 'SEA';

var indexed = _.indexBy(airports, 'code');
console.log(indexed);
/*
{
  STL: {code: 'STL', city: 'St Louis', timeZone: '-6:00'},
  SEA: {code: 'SEA', city: 'Seattle', timeZone: '-8:00'},
  JFK: {code: 'JFK', city: 'New York', timeZone: '-5:00'}
}
*/

var timeZone = indexed[selected].timeZone;
console.log(timeZone);
// -8:00
```

`indexBy()` 函数与 `groupBy()` 函数的运行方式类似，除了每个对象都有唯一的值用来对属性进行索引。所以函数最终得到的结果为一个对象，结果对象的键（该键必须唯一）为每个特定属性对象的值，结果对象的值是处理每个属性的对象。清单 16-6 中，已索引对象的键是每个机场编码，值是相应的机场对象。保证索引对象在内存中具有相对稳定的引用数据是基本的缓存实践。它引入单次性能开销（加下标处理），避免多次迭代开销（每当有某对象需要，都要遍历一次数组）。

16.3 选择

开发者经常会从集合及对象中提取想要的的数据，或删除不想要的的数据。以下几种工作会用到这种数据处理：增强数据的易读性（当数据被显示给用户时）、提高性能（当数据要通过网络连接发送出去）、或隐私保护（当需要从对象或模块 API 中返回稀疏数据）。

16.3.1 从集合中选择数据

Underscore 具有一些列工具函数，这些函数基于某种准则从对象集合中选择一个或多个元素。某些情况下，这个准则可以是一个函数，该函数会计算每个元素的值，然后返回 `true` 或 `false`（根据元素是否“通过了”该准则的测试）；也可以是另外的情况：这个准则是一些数据，集合中的每个（或者部分）元素要和这些数据进行对比判断其是否相等，判定结果的成功或失败决定了元素是否与该准则“匹配”。

1. filter()

filter() 函数使用上述准则中的判别函数的方法。给定一个元素数组及一个函数，**filter()** 为每个元素应用判别函数，然后只返回那些通过判别测试的元素。清单 16-7 中为扑克牌数组应用 **filter** 函数，结果数组中则只返回黑桃扑克牌。

清单 16-7 用判别函数过滤数组

```
// example-007/index.js
'use strict';
var _ = require('underscore');

var cards = [
  {suite: 'Spades', denomination: 'King'},
  {suite: 'Hearts', denomination: '10'},
  {suite: 'Clubs', denomination: 'Ace'},
  {suite: 'Spades', denomination: 'Ace'},
];

var filtered = _.filter(cards, function (card) {
  return card.suite === 'Spades';
});

console.log(filtered);
/*
[
  { suite: 'Spades', denomination: 'King' },
  { suite: 'Spades', denomination: 'Ace' }
]
*/
```

2. where()

where() 函数与 **filter()** 函数相似，只是其使用的判定方法为比较法。其第一个参数是一个对象数组，第二个参数是一个准则对象，其键和值可以和数组中的每个元素进行比较。若某个元素包含准则对象中的所有键和对应的值，元素将会被包含在 **where()** 函数返回的对象中。清单 16-8 展示了一个由准则对象过滤的棋盘游戏对象，该对象中指定了最小玩家数量和游戏时间。结果中 **title** 值为 **Pandemic** 的对象被排除。尽管它的 **minPlayer** 值是与准则对象匹配的，但它的 **playTime** 值不匹配。

清单 16-8 用比较准则过滤数组

```
// example-008/index.js
'use strict';
var _ = require('underscore');

var boardGames = [
  {title: 'Ticket to Ride', minPlayers: 2, playTime: 45},
  {title: 'Pandemic', minPlayers: 2, playTime: 60},
  {title: 'Munchkin Deluxe', minPlayers: 2, playTime: 45}
];

var filtered = _.where(boardGames, {
  minPlayers: 2,
  playTime: 45
});

console.log(filtered);
/*
```



```
[
  { title: 'Ticket to Ride', minPlayers: 2, playTime: 45 },
  { title: 'Munchkin Deluxe', minPlayers: 2, playTime: 45 }
]
*/
```

3. find() 和 findWhere()

上文中的 `filter()` 和 `where()` 函数总是返回集合。若没有任何对象通过准则函数测试，最终就返回空集合。开发者尽管可以使用这些函数查找集合中的某个对象（比如通过某种唯一标识符），但最后还需要通过查找结果数组中第 0 个元素来获取该对象。幸运的是，Underscore 提供 `find()` 和 `findWhere()` 函数，用于补充 `filter()` 和 `where()`。它们都返回第一个通过了判别检验的对象，或返回 `undefined`——若通过判定的结果集为空。清单 16-9 中，为了得到特定词目，集合被搜索了两次。注意，尽管多个条目都能满足传递给 `findWhere()` 的判定对象 `{what: 'Dagger'}`，仅仅是集合中的第一次匹配成功的对象会返回。

清单 16-9 集合中查找单一项目

```
// example-009/index.js
'use strict';
var _ = require('underscore');

var guesses = [
  {who: 'Mrs. Peacock', where: 'Lounge', what: 'Revolver'},
  {who: 'Professor Plum', where: 'Study', what: 'Dagger'},
  {who: 'Miss Scarlet', where: 'Ballroom', what: 'Candlestick'},
  {who: 'Reverend Green', where: 'Conservatory', what: 'Dagger'}
];

var result = _.find(guesses, function (guess) {
  return guess.where === 'Ballroom';
});

console.log(result);
// { who: 'Miss Scarlet', where: 'Ballroom', what: 'Candlestick' }

result = _.findWhere(guesses, {what: 'Dagger'});

console.log(result);
// { who: 'Professor Plum', where: 'Study', what: 'Dagger' }
```

16.3.2 从对象中选择数据

到目前为止，Underscore 中的函数覆盖所有的过滤器。当集合数据中某一部分不需要时，这些过滤器会把较大的集合过滤为集中的、较小的集合（甚至单个对象）。对象也是数据的集合，只是其索引是字符串键值而不是有序的数字；如同数组，在单一对象中过滤数据会很有用。

1. pluck()

通过循环集合中每个元素，或通过使用 `Array.prototype.map()`（或 Underscore 中同等的 `map()` 函数），开发者能够从集合中获取每个对象的属性值，以及从数组中捕获想要的属性值。但更快速、更便捷的方式是使用 Underscore 的 `pluck()` 函数。它的第一个参数是一个数组，第二个参数是每个元素中的属性名。在清单 16-10 中，`pluck()` 函数用于获取落地后的三个色

子朝上的数字。这些数值随后相加（通过 `Array.prototype.reduce()` 相加），决定掷出色子的总值。

清单 16-10 从集合中获取去掉属性

```
// example-010/index.js
'use strict';
var _ = require('underscore');

var diceRoll = [
  {sides: 6, up: 3},
  {sides: 6, up: 1},
  {sides: 6, up: 5}
];

var allUps = _.pluck(diceRoll, 'up');

console.log(allUps);
// [ 3, 1, 5 ]

var total = allUps.reduce(function (prev, next) {
  return prev + next;
}, 0);

console.log(total);
// 9
```

从对象中选取属性时，`pluck()` 是很有用的。它仅在集合上起作用，对处理单个对象则用处不大。

2. values()

ECMAScript 5 标准在 `Object` 构造函数上加入了 `keys()` 函数。它是个很方便的工具，用于把对象字面量的键转换为字符串数组。Underscore 不仅实现了与之对应的 `keys()` 函数，而且实现了原生 JavaScript 所没有的 `values()` 函数。`values()` 函数用于提取对象中所有属性的值，并且可以说对持有常量集合的对象来说大概是最有价值的（父亲的冷笑话），或充当在其他语言中枚举的作用。清单 16-11 展示了这种提取是如何做的。

清单 16-11 获取对象字面量上的值

```
// example-011/index.js
'use strict';
var _ = require('underscore');

var BOARD_TILES = {
  IND_AVE: 'Indiana Avenue',
  BOARDWALK: 'Boardwalk',
  MARV_GARD: 'Marvin Gardens',
  PK_PLACE: 'Park Place'
};

var propertyNames = _.values(BOARD_TILES);

console.log(propertyNames);
// [ 'Indiana Avenue', 'Boardwalk', 'Marvin Gardens', 'Park Place' ]
```

引用数据（如美国各州的缩写及全称之间的哈希）经常会被一次性获取全部并立刻缓存下来。

这类数据通常会通过键来解引用，用来提取某些特殊值。但有时需要处理那类无关乎键的数据，`values()`就很实用了，正如清单 16-12 所示的 Underscore 模板。（Underscore 模板在本章的后续内容会做讨论，不过清单 16-12 中的示例已经展示了基本用法。）散列表 `BOARD_TILES` 中的每个值（街道贴片）都被渲染为 html 中无序列表的一个列表项。本例中数据的键没什么意义；只有值比较重要，可以说是使用 `values()` 函数的完美场景。

清单 16-12 获取对象字面量上的值

```
<!-- example-011/index.html -->
```

```
<div id="output"></div>
```

```
<script id="tiles-template" type="text/x-template">
```

```
<ul class="properties">
```

```
<% _.each(_.values(tiles), function (property) { %>
```

```
<li><%- property %></li>
```

```
<% }); %>
```

```
</ul>
```

```
</script>
```

```
<script>
```

```
(function (_) {
```

```
var template = document.querySelector('#tiles-template').innerHTML;
```

```
var bindTemplate = _.template(template);
```

```
var BOARD_TILES = {
```

```
IND_AVE: 'Indiana Avenue',
```

```
BOARDWALK: 'Boardwalk',
```

```
MARV_GARD: 'Marvin Gardens',
```

```
PK_PLACE: 'Park Place'
```

```
};
```

```
var markup = bindTemplate({tiles: BOARD_TILES});
```

```
document.querySelector('#output').innerHTML = markup;
```

```
})(window._);
```

```
</script>
```

3. pick()

最后，开发者要想把对象削减为其键值对的子集，可以使用 Underscore 的 `pick()` 函数。向 `pick` 函数传递目标对象及一个或多个属性名时，`pick` 函数会返回另一个对象。该对象由目标对象的部分属性（及对应的值）组成，这部分属性也就是 `pick` 函数参数中指定的属性。

清单 16-13 从字面量对象中选取属性

```
// example-012/index.js
```

```
'use strict';
```

```
var _ = require('underscore');
```

```
var boardGame = {
```

```
name: 'Settlers of Catan',
```

```
designer: 'Klaus Teuber',
```

```
numPlayers: [3, 4],
```

```
yearPublished: 1995,
```

```
ages: '10+',
```

```
playTime: '90min',
```

```
subdomain: ['Family', 'Strategy'],
```

```
category: ['Civilization', 'Negotiation'],
```

```
website: 'http://www.catan.com'
```

```
};
```

```
var picked = _.pick(boardGame, 'name', 'numPlayers');
```

```
console.log(picked);
```

```
/*
{
  name: 'Settlers of Catan',
  numPlayers: [ 3, 4 ]
}
*/
```

4. omit()

`pick()`函数倒过来就是 `omit()`函数。`omit()`函数返回这样一个对象：它由所有除了参数中指定参数以外的属性组成。清单 16-14 中，通过 `omit` 函数，属性 `designer`、`numPlayers`、`yearPublished`、`ages` 和 `playTime` 都在结果对象中被剔除。

清单 16-14 从对象字面量中删除属性

```
// example-013/index.js
'use strict';
var _ = require('underscore');

var boardGame = {
  name: 'Settlers of Catan',
  designer: 'Klaus Teuber',
  numPlayers: [3, 4],
  yearPublished: 1995,
  ages: '10+',
  playTime: '90min',
  subdomain: ['Family', 'Strategy'],
  category: ['Civilization', 'Negotiation'],
  website: 'http://www.catan.com'
};

var omitted = _.omit(boardGame, 'designer', 'numPlayers',
  'yearPublished', 'ages', 'playTime');

console.log(omitted);
/*
{
  name: 'Settlers of Catan',
  subdomain: [ 'Family', 'Strategy' ],
  category: [ 'Civilization', 'Negotiation' ],
  website: 'http://www.catan.com'
}
*/
```

除了属性名以外，`pick()`函数和 `omit()`函数接受断言，该断言能够对每个属性和值进行求值。如果断言返回真，该属性就会包含在结果对象中；如果返回假，该属性就被排除。清单 16-15 中，`pick()`的断言让结果对象只包含属性值为数组的元素；本例结果对象包含的属性即 `numPlayers`、`subdomain` 和 `category`。

清单 16-15 使用断言函数从对象字面量中获取属性

```
// example-014/index.js
'use strict';
var _ = require('underscore');

var boardGame = {
  name: 'Settlers of Catan',
  designer: 'Klaus Teuber',
```



```

numPlayers: [3, 4],
yearPublished: 1995,
ages: '10+',
playTime: '90min',
subdomain: ['Family', 'Strategy'],
category: ['Civilization', 'Negotiation'],
website: 'http://www.catan.com'
});

var picked = _.pick(boardGame, function (value, key, object) {
  return Array.isArray(value);
});

console.log(picked);
/*
{
  numPlayers: [ 3, 4 ],
  subdomain: [ 'Family', 'Strategy' ],
  category: [ 'Civilization', 'Negotiation' ]
}
*/

```

16.4 链式调用

Underscore 包含若干工具函数，它们时常被一起使用，作为转换数据的管线。向 Underscore 的 `chain()` 函数传递对象或集合后，链式调用就开始了。`chain()` 函数返回一个链式包装，这个包装使许多 Underscore 函数可以流畅地被调用，每次调用都结合了前序函数调用的效果。清单 16-16 展示了一个咖啡店和咖啡店营业时间的数组。`whatIsOpen()` 函数接受数字类型的小时数和时间段（'AM' 或 'PM'）。函数接下来会计算集合中的咖啡店对象，并返回在该时间段营业的咖啡店名字。

清单 16-16 对一个集合进行函数链式调用

```

// example-015/index.js
'use strict';
var _ = require('lodash');

/*
  注意本例中使用了 lodash，而不是 underscore。下面的 cloneDeep() 函数是 lodash 所特有的。
*/

var coffeeShops = [
  {name: 'Crooked Tree', hours: [6, 22]},
  {name: 'Picasso's Coffee House', hours: [6, 24]},
  {name: 'Sump Coffee', hours: [9, 16]}
];

function whatIsOpen(hour, period) {
  return _.chain(coffeeShops)
    .cloneDeep() // #1
    .map(function to12HourFormat (shop) { // #2
      shop.hours = _.map(shop.hours, function (hour) {
        return (hour > 12 ? hour - 12 : hour);
      });
      return shop;
    })
    .filter(function filterByHour (shop) { // #3
      if (period === 'AM') {

```

```

    return shop.hours[0] <= hour;
  }
  if (period === 'PM') {
    return shop.hours[1] >= hour;
  }
  return false;
})
.map(function toShopName (shop) { // #4
  return shop.name;
})
.value(); // #5
}
console.log(whatIsOpen(8, 'AM'));
// [ 'Crooked Tree', 'Picasso's Coffee House' ]

console.log(whatIsOpen(11, 'PM'));
// [ 'Picasso's Coffee House' ]

```

`coffeeShops` 包装一层 `chain()` 函数后形成了流畅的 API。如下函数被依次调用，对集合进行筛选，直到生成期望的数据。

1. `cloneDeep()` 函数递归的复制数组和所有对象及其属性。由于在步骤 2 中，数组数据实际上会被修改，因此要复制该数组来保存其原始状态。

2. `map(function to12HourFormat(){/*...*/})` 操作迭代循环了克隆后数组中每个元素并把时间由 24 时制转换为 12 时制。

3. `filter(function filterByHour(){/*...*/})` 迭代了每个修改后的咖啡店元素，并基于指定的('AM'或'PM')时间段计算了咖啡店营业小时数：第一个元素用于营业时间，第二个元素用于关店时间。函数返回真或假，表示该咖啡店元素是否需要在结果中保留。

4. `map(function toShopName(){/*...*/})` 返回集合中剩下的咖啡店名称。结果是字符串数组可以作为参数传进链式调用的任意后续步骤中。

5. 最终，链式操作以 `value()` 函数的调用终止并返回最终结果：一个咖啡店的名字字符串数组，其中的咖啡店满足传入 `whatIsOpen()` 函数中营业时间段（如果没有一个元素匹配该准则，则返回空数组）。

看起来有很多内容需要领悟，不过 Underscore 链可以化简为几个简单易记的原则：

- 链式操作可以由任意初始值开始——尽管最典型的用法是把对象或数组作为起始点。
- 任意 Underscore 操作数值的函数都可以用作链式操作中的函数。
- 链式操作中前一个函数的返回值是下一个函数的输入值。
- 链式函数的第一个参数总是该函数所需操作的值。例如，Underscore's `map()` 函数通常接受两个参数：一个集合和一个回调函数。但当函数以链式调用，它则只接受一个回调函数。此模式适用于所有链式函数。
- 通常，通过调用 `value()` 函数来终止一次链式调用并得到最终操作后的值。若一次链式操作没有返回一个值，那么这次操作就是不必要的。

尽管对集合或对象应用链式调用也许看起来显而易见的自然，但 Underscore 同样有很多用在原始类型上的函数。清单 16-17 显示了数字 100 如何经过链式调用的包装最终生成歌词“99 Bottles of Beer”。

清单 16-17 原始类型值上的链式调用

```

// example-016/index.js
'use strict';
var _ = require('underscore');

_.chain(100)

```

```

.times(function makeLyrics (number) {
  if (number === 0) {
    return '';
  }
  return [
    number + ' bottles of beer on the wall!',
    number + ' bottles of beer!',
    'Take one down, pass it around!',
    (number - 1) + ' bottles of beer on the wall!',
    '♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪',
  ].join('\n');
})
.tap(function orderLyrics (lyrics) {
  // 翻转数组使歌词有序
  lyrics.reverse();
})
.map(function makeLoud (lyric) {
  return lyric.toUpperCase();
})
.forEach(function printLyrics (lyric) {
  console.log(lyric);
});

```

`times()` 函数将数字作为第一个参数，以及一个回调函数将调用每个递减数字的值。在本例中，回调函数 `makeLyrics()` 从数字 99（而不是 100）被开始调用，并以数字 0 结束调用，共计迭代 100 次。每次调用就会返回一句“99 Bottles”叠句。它创建了一个字符串数组，该数组紧接着传递到链式调用的下一个函数中。因为链式调用的最后一个函数 `forEach()` 没有返回一个值，所以产生了副作用，此时没有必要使用 `value()` 函数终止链式调用。相反，清单 16-18 显示出结果被打印到控制台上。

清单 16-18 毁掉整个自驾游之歌

```

99 BOTTLES OF BEER ON THE WALL!
99 BOTTLES OF BEER!
TAKE ONE DOWN, PASS IT AROUND!
98 BOTTLES OF BEER ON THE WALL!
♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪
98 BOTTLES OF BEER ON THE WALL!
98 BOTTLES OF BEER!
TAKE ONE DOWN, PASS IT AROUND!
97 BOTTLES OF BEER ON THE WALL!
♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪

```

...

16.5 函数计时

函数从排在 JavaScript 的内部事件循环中后开始执行。原生函数如 `setTimeout()`、`setInterval()` 以及 Node 的 `setImmediate()`，让开发者在函数运行时能一定程度上控制事件循环处理函数调用的次序。在函数调度上，Underscore 提供了很多函数以增强原生函数的灵活性。

16.5.1 defer()

Underscore 的 `defer()` 模拟了 Node.js 环境中的 `setImmediate()` 行为；也就是说，`defer()` 把

函数安排在事件循环的下一个立即调用次序中执行，相当于以延迟数 0 执行 `setTimeout()`。由于 `setImmediate()` 并不是 JavaScript 标准函数，比起浏览器中的臆子函数 `setImmediate()`，Underscore 的 `defer()` 函数在浏览器和服务端环境中都有更程度的一致性。

清单 16-19 中的示例代码展示了用户界面中 `defer()` 求得的值。该程序载入了一个较大的数据集，集合中记录了流行卡牌游戏 `dominion` 的玩牌信息，随后通过卡牌详情数据生成 HTML 中表格元素。

当数据从服务端获取并处理后，用户就能看到消息：“卡牌加载中请耐心等待！”一旦 GET 请求完成，`processCards()` 函数开始以每次处理 10 张的方式处理大约 200 张卡牌。对于每次处理（第一次除外），处理函数都延迟执行，其具有两点好处：第一，它允许 UI 绘制处理过的前十行元素来绘制表格；第二，它允许用户在视窗重绘之间滚动。由于每次处理的元素很少，滚动速度对用户来说也相对正常。如果不使用这种方式而使 `processCards()` 立刻渲染表格中所有列，UI 就会被冻结，直到所有 DOM 元素都添加到表格中。

清单 16-19 函数延迟执行

```
<!-- example-017/views/defer.html -->
<p id="wait-msg">Please be patient while cards are loading!</p>
<table id="cards">
  <thead>
    <tr>
      <th>Name</th>
      <th>Expansion</th>
      <th>Cost</th>
      <th>Benefit</th>
      <th>Description</th>
    </tr>
  </thead>
  <tbody></tbody>
</table>

<script>
$(function () {
  var $waitMsg = $('#wait-msg');
  var $cards = $('#cards tbody');

  function processCards(cards) {
    var BLOCK_SIZE = 10;

    // 处理第一个数据块中的 10 张卡牌
    (function processBlock() {
      if (!cards.length) {
        $waitMsg.addClass('hidden');
        return;
      }

      // 从数组中获取前 10 张卡牌数据;
      // splice() 操作将数组的长度每次减 10
      var block = cards.splice(0, BLOCK_SIZE);

      $.each(block, function (card) {
        var $tr = $('<tr></tr>');
        $tr.append($('<td></td>').html(card.name));
        $tr.append($('<td></td>').html(card.expansion));
        $tr.append($('<td></td>').html(card.cost));
        $tr.append($('<td></td>').html(card.benefits.join(', ')));
        $tr.append($('<td></td>').html(card.description));
        $cards.append($tr);
      });
    })();
  }
});
```



```

// 延迟处理下一个卡牌数为 10 的数据块，以允许用户滚动页面刷新 UI
.defer(processBlock);
})();
}

// 通过加载数据集开始处理过程
$.get('/cards').then(processCards);
})();
</script>

```

16.5.2 debounce()

“防抖动”是一种用于在系统中一段时间内忽略重复调用、请求、消息等的实践。在 JavaScript 中，如果开发者预计重复的、完全相同的函数会被快速连续地调用，去除函数抖动会变得非常有用。例如，一个常见情况是用户在网页上不小心多次单击了提交按钮时，函数防抖能避免表单提交处理函数被多次调用。

实现自定义防抖函数需要跟踪函数在较短时间段内的调用过程（可能只有几百毫秒），对每次重复的调用使用 `setTimeout()` 和 `clearTimeout()`。所幸的是，Underscore 提供了 `debounce()` 函数帮开发者实现这个过程，如清单 16-20 所示。

清单 16-20 去除函数抖动

```

<!-- example-018/debounce.html -->
<button id="submit">Quickly Click Me Many Times!</button>
<script>
(function () {
  var onClick = _.debounce(function (e) {
    alert('click handled!');
  }, 300);

  document.getElementById('submit')
    .addEventListener('click', onClick);
})();
</script>

```

清单 16-20 中的 `onClick()` 函数由 `debounce()` 调用后得到。`debounce()` 第一个参数为函数为函数，一旦所有重复的调用停止后，该函数会执行一次。第二个参数是一个时间段，以毫秒为单位，表示回调函数最终被调用两次之间的时间间隔。例如，用户单击了一次提交按钮，随后在 300 毫秒之内又单击了一次，那么第一次调用会被忽略，等待计时器重新计时。一旦等待时间到了，`debounce()` 的回调函数才被调用，提示用户单击已经被处理。

■ **注意** 每当调用一次防抖函数后，其内部的计时器就会重置。参数中指定的时间段代表后一次调用和前一次调用（若之前存在任意一次调用）必须经过的时间间隔。

图 16-1 中设定了间隔时间为 300 ms 的防抖函数被调用了 3 次。在 A 点处第一次调用之后，经过 250 ms，B 点处再次发生一次调用，等待计时器重置。B 点和下一次调用点 C 点的时间间隔更短一些：100 ms。等待计时器再次重置。在 C 点处发生了第三次调用，之后需要等待 300 ms。D 点处，防抖函数的回调函数调用了。

防抖函数的回调函数接受任意传递给 `debounce()` 函数的参数。例如，清单 16-20 中，jQuery 的事件对象 `e` 发送给防抖函数的回调函数。每次调用会传递不同的参数。必须注意，该参数只能传递

进经过等待时间段后最后一次调用的函数中。`debounce()` 函数接受的第三个参数是可选的，表示是否立即调用。把这个参数设为 `true`，`debounce` 函数就会在一开始调用回调函数，而不是忽略等待时间段中的后续重复函数。若传递给防抖函数的参数发生变化，获取第一个传入的参数而不是最后一个则很有用。

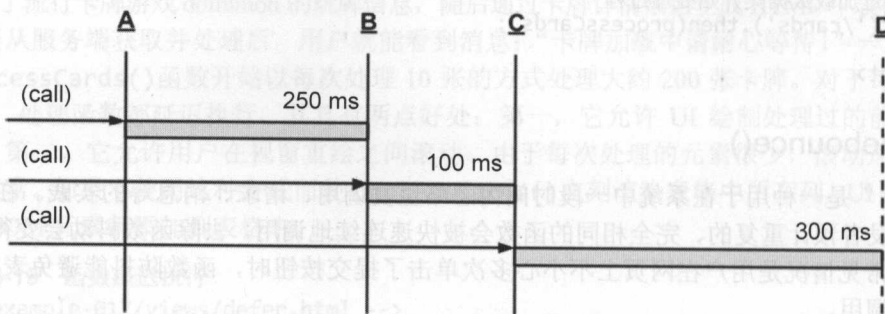


图 16-1 多次调用防抖函数

16.5.3 throttle()

Underscore 的 `throttle()` 函数与 `debounce()` 函数类似，它会忽略某一时间段内后续的函数调用，但每次函数调用不会重置内部的计时器。这有效地保证了在某个时间段内只有唯一一次调用；反之，`debounce()` 函数则保证在上一次调用结束后的某段时间内只发生一次调用。当函数可能会多次以相同参数调用，或者当参数的粒度并不要求每次都被调用时，函数节流相当有用。

对于应用中的路由来说，JavaScript 内存消息总线 `postal.js` 是很常用的库。某些应用模块发送的消息可能很频繁，但其实我们并不需要处理那么多信息。这时，为用户显示消息的函数很适合做节流。清单 16-21 显示了上述场景的简化版，不用太担心如何理解 `postal.js` 的全部 API，理解 `postal.publish()` 会向总线上放一条消息，然后 `postal.subscribe()` 会在消息到达时调用回调就足够了。本例中，消息每 100 ms 发布一次。然而订阅消息时附属回调函数会被节流为 500 ms 一次。因此，为计时不一致性增加少量填充（JavaScript 时间循环计时器精度较低）。尽管消息总线上更新了 100 次消息，但 UI 只会做 20 或 21 次更新（5 次消息大概显示 1 条）。

清单 16-21 为状态控制更新函数做节流

```
<!-- example-019/throttle.html -->
<section id="friends"></section>
```

```
<script>
```

```
$(function () {
```

```
    var $friends = $('#friends');
```

```
    function onStatusUpdate(data) {
        var text = data.name + ' is ' + data.status;
        $friends.append($('

<p></p>').html(text));
    }


```

```
    /*
     * 为朋友频道订阅状态更新，函数节流后，每 500 ms 后调用一次
     */
```

```
    postal.subscribe({
        channel: 'friends',
```

```

topic: 'status.update',
callback: _.throttle(onStatusUpdate, 500)
});

})();
</script>

<script>
$(function () {
  var i = 1;
  var interval = null;

  /*
   * 每 100 ms 从朋友频道更新状态
   */
  function sendMessage() {
    if (i === 100) {
      return clearInterval(interval);
    }
    i += 1;
    postal.publish({
      channel: 'friends',
      topic: 'status.update',
      data: {
        name: 'Jim',
        status: 'slinging code'
      }
    });
  }
  setInterval(sendMessage, 100);
})();
</script>

```

图 16-2 说明了 `throttle()` 与 `defer()` 的不同之处。一旦点 A 处调用一次函数节流，后面的调用将被忽略（B 和 C 点），直到本例中 300 ms 的等待时间结束，下一次调用即 D 点会调用节流后的函数。

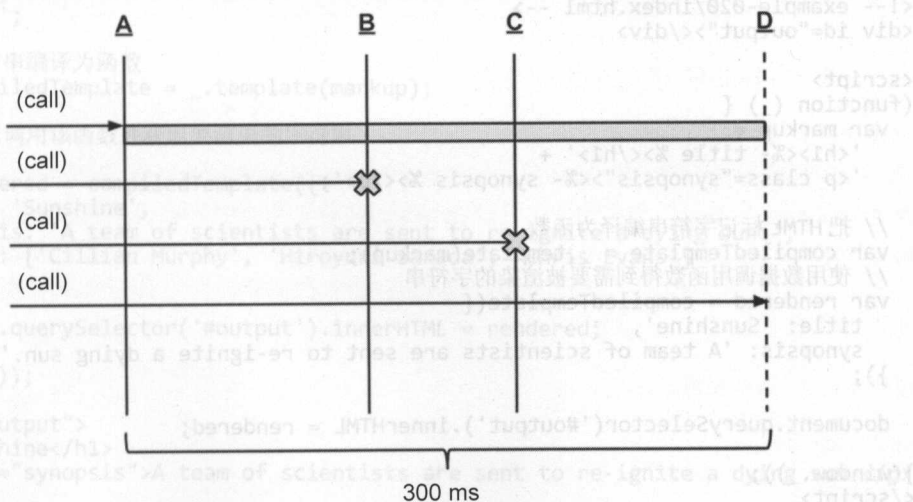


图 16-2 多次调用节流函数

16.6 模板

Underscore 提供了微模板系统，将模板字符串（主要是 HTML）转换为函数。当通过某数据调用函数时，使用模板字符串的绑定表达式计算模板，并返回新的 HTML 字符串。用过类似 Mustache 或 Handlebars 的模板工具的开发者会很熟悉这个过程。不同于那些功能强大的模板库，Underscore 的模板功能更少，也没有真正的模板扩展点。当应用中的模板是相当实验性的，并且你也没有意愿或需求引入一个特定的模板引擎库以至于增大开销，Underscore 的模板库可以说是很不错的选择。模板系统通常以某些标记开始，Underscore 也不例外，数据绑定表达式以“鳄鱼标签”（之所以这样称呼，是因为开始标记`<%`和结束标记`%>`看起来有点像短吻鳄）。清单 16-22 显示了一块简单的 HTML 标记代码，随后会绑定到对象字面量上。该对象字面量包含两个属性：标题和大纲。

清单 16-22 带有鳄鱼标签的模板

```
<h1><%- title %></h1>
<p class="synopsis"><%- synopsis %></p>
```

鳄鱼标签有三种。通过把所有 HTML 标签序列转译，清单 16-22 中使用的标签输出安全的 HTML 代码。如果电影概要包含一个 HTML 标签，这个标签就会被转换为``。相比之下，鳄鱼标签`<%`可以用来输出不经过转码原封不动的 HTML 标记。第三个鳄鱼标签是 JavaScript 求值标签，它简单地以`<%`开始（更多关于这个标签的细节都会被提及）。所有鳄鱼标签的结束标记都是同样的`%>`。把清单 16-22 所示的 HTML 转换为填充后的模板，首先要通过把 HTML 字符串传入 Underscore 的 `template()` 函数中编译，随后返回可重用的绑定函数。当数据对象传递到绑定函数中后，原始模板字符串中任何与绑定表达式匹配的属性最终都被替代，计算得到输出。Underscore 对其内部实现使用了 JavaScript 的 `with` 关键字，魔法般地把这些属性加入到模板作用域中。清单 16-23 显示了如何把一个简单的模板字符串与数据对象绑定，也显示了最终结果输出的 HTML 代码。

清单 16-23 绑定 Underscore 模板

```
<!-- example-020/index.html -->
<div id="output"></div>

<script>
(function (_) {
  var markup =
    '<h1><%- title %></h1>' +
    '<p class="synopsis"><%- synopsis %></p>';

  // 把 HTML 标记字符串编译为函数
  var compiledTemplate = _.template(markup);
  // 使用数据调用函数得到需要被渲染的字符串
  var rendered = compiledTemplate({
    title: 'Sunshine',
    synopsis: 'A team of scientists are sent to re-ignite a dying sun.'
  });

  document.querySelector('#output').innerHTML = rendered;
})(window._);
</script>
<div id="output">
  <h1>Sunshine</h1>
```



```
<p class="synopsis">A team of scientists are sent to re-ignite a dying sun.</p>
</div>
```

一旦模板字符串编译为函数，它可以以不同的数据被多次调用，从而生成不同的要被渲染的标记。在页面加载期间（或在 Node.js 运行环境下，应用程序启动时），应用程序把模板字符串编译为函数是种普遍做法，随后在整个应用程序生命周期中都可以随需要调用此函数。若模板字符串不会改变，也就没必要重新编译它们。

16.6.1 模板内的循环及其他 JavaScript 代码

处理烦琐的模板工作，比如迭代集合时，很多模板库都包含缩略记法。为保证模板系统的轻巧，Underscore 定制了语法糖，让开发者能用简洁有效的 JavaScript 代码书写模板。

清单 16-24 中，在模板内使用 Underscore 的 `each()` 函数，就能够创建演员数据的无序列表。这里需要注意两个重点：第一，JavaScript 代码在鳄鱼标记的代码块中执行。创建这种代码块的鳄鱼标记的开始标记（如 `<% %>`，而不是 `<%- %>`）中不带连字符。第二，`each()` 循环在中间被分割，即由循环创建的列表元素中，也就是用来渲染变量“演员”的那段模板标记。最后，像平常的 JavaScript 循环书写方式一样，以右大括号、右括号、分号结束。

清单 16-24 模板中的循环操作

```
<!-- example-021/index.html -->
<div id="output"></div>

<script>
(function (_) {
  var markup =
    '<h1><%- title %></h1>' +
    '<p class="synopsis"><%- synopsis %></p>' +
    '<ul>' +
    '<%.each(actors, function (actor) { %>' +
    '<li><%- actor %></li>' +
    '<% %>'; %>' +
    '</ul>';

  // 把字符串编译为函数
  var compiledTemplate = _.template(markup);

  // 用数据调用该函数并获取要渲染的字符串

  var rendered = compiledTemplate({
    title: 'Sunshine',
    synopsis: 'A team of scientists are sent to re-ignite a dying sun.',
    actors: ['Cillian Murphy', 'Hiroyuki Sanada', 'Chris Evans']
  });

  document.querySelector('#output').innerHTML = rendered;
})(window._);
</script>
<div id="output">
  <h1>Sunshine</h1>
  <p class="synopsis">A team of scientists are sent to re-ignite a dying sun.</p>
  <ul>
    <li>Cillian Murphy</li>
    <li>Hiroyuki Sanada</li>
    <li>Chris Evans</li>
```

```
</ul>
</div>
```

JavaScript 执行标签同样可以用来执行任意 JavaScript 代码。清单 16-25 中的模板为电影数据计算了已投票人数与总关注人数的百分比。模板使用 Underscore 内部的 `print()` 函数来渲染计算输出的模板结果，在表达式更复杂的情况下，这种方式有时可以用于替代鳄鱼标签。

清单 16-25 模板内执行任意 JavaScript 代码

```
<!-- example-022/index.html -->
<div id="output"></div>

<script>
(function (_) {
  var markup =
    '<p>' +
    '<%- voted %> out of <%- total %> stars!' +
    ' (<% print((voted / total * 100).toFixed(0)) %>%)' +
    '</p>';

  var compiledTemplate = _.template(markup);

  var rendered = compiledTemplate({
    voted: 4, total: 5
  });

  document.querySelector('#output').innerHTML = rendered;
})(window._);
</script>
<div id="output">
  <p>4 out of 5 stars! (80%)</p>
</div>
```

注意 通常在模板中（也就是应用的视图层中）执行计算操作是很糟糕的实践。相反，传递给编译后模板函数的数据应该由已经计算过的值组成。清单 16-25 中的用法仅仅作为演示使用。

16.6.2 书写不加鳄鱼标记的代码

在重要模板中，鳄鱼标签显得有些不太容易写。幸运的是，在 Underscore 中，开发者可以用正则表达式改变模板标签的语法。为 Underscore 对象中的 `templateSettings` 设置一个新的哈希键值对，可以改变 Underscore 在页面生命周期中（或在 Node.js 进程中）的行为，同时会影响所有模板渲染。

清单 16-26 显示了如何改变 Underscore 的鳄鱼标记语法，使之变为更简洁的 Mustache/ Handlebars 语法形式。本例中，三种不同类型的标签（求值、插值和转译插值）都在全局设置对象中被赋值了相应的正则表达式。

清单 16-26 改变模板语法

```
<!-- example-023/index.html -->
<div id="output"></div>

<script>
(function (_) {
  _.templateSettings = {
    // 任意 JavaScript 代码块: {{ }}
  }
})
```

```

evaluate: /\{\{(.+?)\}\}/g,
// 不安全的字符插值: {\{= \}\}
interpolate: /\{\{=(.+?)\}\}/g,
// 转译后的字符插值: {\{- \}\}
escape: /\{\{-(.+?)\}\}/g
};
var markup =
  '<h1>{\{- title \}}</h1>' +
  '<p class="synopsis">{\{- synopsis \}}</p>' +
  '<ul>' +
  '{\{ .each(actors, function (actor) { \}}' +
  '<li>{\{- actor \}}</li>' +
  '{\{ \}}; \}}' +
  '</ul>';

var compiledTemplate = _.template(markup);
var rendered = compiledTemplate({
  title: 'Sunshine',
  synopsis: 'A team of scientists are sent to re-ignite a dying sun.',
  actors: ['Cillian Murphy', 'Hiroyuki Sanada', 'Chris Evans']
});

document.querySelector('#output').innerHTML = rendered;
})(window._));
</script>

```

例子中任何由模板系统编译后的标记目前必须支持指定的 Mustache 语法。依然使用鳄鱼标签的模板字符串则不能被正确渲染。

表 16-2 可以作为便捷参考，其中列出了模板设置及其语法和相应的正则表达式。

表 16-2 全局模板设置

设置	模板语法	正则表达式
evaluate	{\{ ... \}}	/\{\{(.+?)\}\}/g
interpolate	{\{= ... \}}	/\{\{=(.+?)\}\}/g
escape	{\{- ... \}}	/\{\{-(.+?)\}\}/g

16.6.3 从模板中获取数据对象

如前所述，Underscore 在模板的作用域中使用 JavaScript 的 `with` 关键字作为“头等公民”变量，计算数据对象的属性。对象自身同样会通过模板中的 `obj` 属性引用。清单 16-27 中的代码对前面的例子稍微修改，模板在计算百分比之前，在 `if/else` 代码段中对数据属性 `obj.percent` 进行检测。若数据对象中具有 `percent` 属性，该属性值就会被渲染；否则，渲染计算后的值。

清单 16-27 “obj”变量

```

<!-- example-024/index.html -->
<div id="output"></div>

<script>
(function (_) {
  var markup =
    '<%- voted %> out of <%- total %> stars!' +
    '<% if (obj.percent) { %>' +
    ' (<%- obj.percent %>)' +

```

```
'<% } else { %>' +
'(<% print((voted / total * 100).toFixed(0)) %>)' +
'<% } %>;

var compiledTemplate = _.template(markup);

var rendered = compiledTemplate({
  voted: 4, total: 5, percent: 80.2
});

document.querySelector('#output').innerHTML = rendered;

}(window._));
</script>
```

作为一种细微优化（也为了安全特性）手段，给作用域对象起一个名字，就可以完全不需要使用 `with` 关键字。这让模板函数运行得更快一些，也让模板中所有属性能够直接通过命名数据对象的属性来引用。给数据对象指定名字，需要在模板编译阶段，将对象传入 Underscore 的 `template()` 函数内。该对象的 `variable` 属性值会作为数据对象的变量名，使该名字之后可以直接在模板中引用。清单 16-28 显示了实践中如何做上述设置。

清单 16-28 为数据对象的变量名

```
<!-- example-025/index.html -->
<div id="output"></div>

<script>
(function (_) {
  var markup =
    '<%- movie.voted %> out of <%- movie.total %> stars!' +
    '<% if (movie.percent) { %>' +
    '(<%- movie.percent %>)' +
    '<% } else { %>' +
    '(<% print((movie.voted / movie.total * 100).toFixed(0)) %>)' +
    '<% } %>';

  var settings = {variable: 'movie'};
  // settings 是第三个参数
  var compiledTemplate = _.template(markup, null, settings);

  var rendered = compiledTemplate({
    voted: 4, total: 5, percent: 80.1
  });

  document.querySelector('#output').innerHTML = rendered;

}(window._));
</script>
```

注意 `variable` 属性可以设置在 Underscore 的全局设置中。然而，为变量设置又好又合适的名字很重要，根据上下文给变量命名会更合理。不同于定义某些通用变量名如 `data` 或 `item`，本例中使用的名字为 `movie`。当编译电影模板时，把该配置对象传递到 `template()` 中。

16.6.4 默认模板数据

尽管 Underscore 的 `defaults()` 函数不是模板系统的一部分，但可用来保证模板总会有默认数据。当数据对象缺少一个或多个属性时，此函数可以避免数据绑定失败。`defaults()` 函数的第一个

参数是可能缺失了部分属性的对象。任何后续参数则是设置了默认值的对象，用于填充第一个参数中对象缺失的属性值。返回值对象代表所有参数的属性“合并后”的对象。清单 16-29 显示了数据对象上缺失 `synopsis` 属性值的效果。在 `data` 对象和 `DEFAULTS` 对象传入了 `defaults()` 函数后，返回值对象包含来自于 `data` 中的 `title` 和 `DEFAULTS` 中的 `synopsis` 属性。

清单 16-29 默认模板值

```
<!-- example-026/index.html -->
<div id="output"></div>

<script>
(function (_) {
  var markup =
    '<h1><%- title %></h1>' +
    '<p class="synopsis"><%- synopsis %></p>';

  // 把字符串编译为函数
  var compiledTemplate = _.template(markup);

  var DEFAULTS = {
    title: 'A Great Film',
    synopsis: 'An epic hero defeats and evil villain and saves the world!'
  };

  var data = {
    title: 'Lord of the Rings'
  };

  // 使用 defaults 函数填充 data 中任意缺少的值
  var merged = _.defaults(data, DEFAULTS);

  var rendered = compiledTemplate(merged);

  document.querySelector('#output').innerHTML = rendered;

})(window._);
</script>
```

如果传递了多个默认对象到 `defaults()` 中，它们会从第一个到最后一个计算。一旦默认对象中找到第一个缺失属性，后续的任意默认对象就会忽略该属性。

16.7 小结

在原生类型如 `String`、`Array`、`Object`、`Function` 等方面，ECMAScript 的当前及未来的实现给开发者带来很多实用函数。不幸的是，世界变化的速度比标准实现的速度要快，所以像 `Underscore` 和 `Lodash` 这样的函数库满足了开发者的需求，也满足了语言的成熟性。

`Underscore` 囊括超过 100 个工具函数及一个微模板引擎，允许开发者操作、转换、在集合和对象中渲染数据。可以直接在浏览器和服务器环境中使用 `Underscore`，并且不需要任何依赖。它可以通过简单的 `script` 标签或通过 AMD 和 CommonJS 模块引入。开发者对不同的平台，都能从流行的包管理器如 `Bower`、`NPM`、`Component` 和 `NuGet`，下载预编译的 `Underscore` 包。对于 JavaScript 工程来说，`Underscore` 因其强大的特性集和广泛使用的特点，可以比喻为一把完美而不引人注目的瑞士军刀。

16.8 相关资源

- Underscore: <http://underscorejs.org/>
- Lodash: <https://lodash.com/>

欢迎来到异步社区！

异步社区的来历

异步社区 (www.epubit.com.cn) 是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 = 1 元，购买图书时，在 使用积分 里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本书的读者专享异步社区购书优惠券。

使用方法：注册成为社区用户，在下单购书时输入 **S4XC5** **使用优惠券**，然后点击“使用优惠码”，即可在原折扣基础上享受全单9折优惠。（订单满39元即可使用，本优惠券只可使用一次）

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在这一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信服务号



微信订阅号



官方微博



QQ 群：436746675

社区网址：www.epubit.com.cn

投稿 & 咨询：contact@epubit.com.cn

Apress®

专业人士写给专业人士的书

JavaScript开发框架权威指南

JavaScript开发框架可以说是野蛮生长，发展迅速。在过去几年中，JavaScript开发工具大规模发展并日趋成熟。

本书是JavaScript开发框架的一本前沿学习指南。本书介绍了备受资深开发者关注与支持的库、框架和工具，无论新出现的还是较为成熟的库和框架都有所涉及。本书介绍的工具涵盖了整个开发技术栈，既包括客户端也包括服务端。

本书无法囊括每个JavaScript库的全部知识点，它聚焦于生产环境中一些非常实用的库和框架。在本书中，你将会看到依赖管理工具的详细分析和示例代码、模块化和自动化构建任务的代码实现，创建专门的应用服务器、客户端应用程序架构，实现横向扩展，以及管理不同类型的数据库。

本书介绍的库和框架包括 Bower、Grunt、Yeoman、PM2、RequireJS、Browserify、Knockout、AngularJS、Kraken、Mach、Mongoose、Knex、Bookshelf、Faye、Q、Async.js、Underscore 以及 Lodash。

本书由两位专业人士编写，读者从他们的亲身经历可以学习到大量成功与失败的经验，可以迅速了解通常在 API 文档或 README 中没有明确说明的诸多问题，并可以快速学习如何专业地使用JavaScript库和框架。

 异步社区
人民邮电出版社
www.epubit.com.cn



异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机 / 软件开发 / JavaScript
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-44719-7



9 787115 447197 >

ISBN 978-7-115-44719-7

定价：89.00 元